Department of Computer Science

UNIVERSITY OF BATH

Technical Report

Undergraduate Dissertation: Developing an XCS Framework

Thomas N Longridge

Technical Report 2005-10 **October 2005**

ISSN 1740-9497

Contact Address:
Department of Computer Science
University of Bath
Bath, BA2 7AY
United Kingdom
URL: http://www.cs.bath.ac.uk

ISSN 1740-9497

Developing an XCS Framework

Thomas N Longridge
BSc (Hons) in Computer Science

May 2005

Developing an XCS Framework

Submitted by Thomas N Longridge

Copyright

Declaration

This dissertation is submitted to the University of Bath in accordance with the requirements of the degree of Bachelor of Science in the Department of Computer Science. No portion of the work is this dissertation has been submitted in support of an application for any other degree or qualification of this or any other university of institution of learning. Except where specifically acknowledged, it is the work of the author.

Signed .....

Signed .....

Abstract

XCS is a relatively recent development of Learning Classifier System that has proven to be more effective than previous algorithms. These systems are forms of machine learning algorithm that can be used to provide an agent with the means of evolving "desirable" behaviour within a given environment. Several implementations of the XCS algorithm have been written in programming languages such as C and Java. They are used both as learning tools and to further research in the area. This study documents the development of an XCS implementation in the Python programming language. It aims to provide a more accessible framework for XCS research and development based on the algorithmic description given by Butz & Wilson (2000). Initially a procedural implementation is developed that closely emulates this description. The components are then reconstructed into an object-oriented structure that forms the basis of the framework along with some standard environments and a controller class that controls the whole experiment.

Acknowledgements

I would like to thank Dr Alwyn Barry for his initial concept and his continuing support and enthusiasm for the project. Also many thanks Jan Drugowitsch for his advice and to those who helped proof read the various drafts.

Contents

Chapter 1

Introduction

A *Learning Classifier System (LCS)* (Holland 1986) is a machine learning algorithm inspired by processes in the natural world. In order to obtain "desirable" behaviour in a given environment, the LCS is rewarded for performing an action that is correct (or is punished for undesirable behaviour). This reinforcement learning technique, often used to train animals, drives the LCS to develop a set of internal rules that describe the correct responses to particular environmental conditions. In order to learn from its mistakes and explore new possibilities the LCS employs a *Genetic Algorithm (GA)* that acts on the rule-base in a way that emulates the principles of sexual reproduction, random gene mutation and "survival of the fittest" in the natural world.

These mechanisms make an LCS well suited for use in complex problems where a programmer would not be able to provide a solution methodically either because the domain is not well understood or is too large to be adequately handled. Such applications include data mining (Barry, Holmes & Llora 2004) of areas such as financial markets; and modelling or controlling complex systems such as communication networks (Carse, Fogarty & Munro 1996).

XCS (Wilson 1995) is a form of LCS that was developed to improve on some of the short-comings of Holland's LCS. In particular it aimed to produce systems with an optimal set of rules that accurately describe correct behaviour. Wilson bases the learning phase of XCS on the accuracy of the rules that the system uses, rather than the value of the reward itself.

Numerous implementations of XCS have been developed in a variety of popular programming languages such as C and Java. Due to the repeatative nature of the algorithm the code for these implementations needs to be fast and efficient. However this comes at the cost of obscuring the original algorithm. This means that development of the code by anyone unfamiliar with the specific implemenation is often very difficult and time-consuming.

This project aims to provide an XCS framework primarily designed for both research and development, and educational use. In order to do so, it will use a language more capable of representing the algorithmic processes described by Wilson. As a further tool for education purposes, we also implement, as closely as possible, the algorithmic description of an XCS as given by Butz & Wilson (2000).

The first part of this report provides a background on XCS theory and an analysis of the two main software implementations that are currently in

use. Part II describes the preliminary design processes undertaken during the project; including requirements analysis and the top-level design. The detail of the framework's implementation is described in Part III, giving justification for the various design decisions. Finally, an analysis of the performance of the developed framework is given along with a disscussion on its effectiveness and the potential for future development.

Part I

Background

Chapter 2

Learning Classifier Systems & XCS

2.1 Introduction to Learning Classifier Systems

> "Find a bug in a program, and fix it, and the program will work today. Show the program how to find and fix a bug, and the program will work forever."
>
> – Hearst & Hirsh (2002)

This sentiment goes a long way to describe the rationale for the development of so-called "machine learning" software agents. Give an agent the ability to learn appropriate behaviour from its environment and you remove the limitations of a human programmer's logic. This is especially pertenant to domains that contain vast amounts of data with complex relationships such as data mining, network routing, speech recognition and gaming strategies.

Developing a machine learning agent is by no means an easy task. Many techniques have been developed with no one algorithm proven with universal success. The algorithms can usually be categorised into three groups: supervised learning, unsupervised learning and reinforcement learning. In supervised learning, the agent is given some correct inputs and outputs, then develops a function to produce the correct mapping. Unsupervised learning algorithms are only given inputs which it clusters into groups of similar data to build a model. In reinforcement learning, the agent receives input data, but also a reward relating to the "correctness" of its chosen actions which it uses when selecting actions in the future.

First proposed by Holland in 1986, Learning Classifier Systems (LCS) are one form of reinforcement learning technique in machine learning. In broad terms, an LCS seeks to classify inputs from its environment via a set of rules that determine the correct action to take in the given situation. Figure 2.1 shows the relationship between an LCS and its environment. The LCS can detect particular attributes of the environment using a number of sensors and can perform a finite set of actions on the environment using its effectors. Messages are passed from the sensors to the LCS, in the form of a binary string in which each bit corresponds to the state of a particular sensor. Given this information,

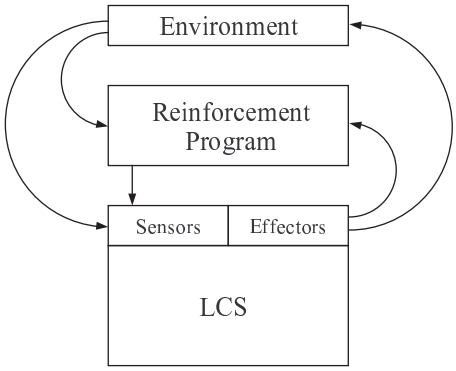

Figure 2.1: The current environment state is determined by the LCS using sensors. Based on this, an action is executed on the environment using the LCS's effectors. Both inputs and outputs are also sent to the Reinforcement Program and it is based on this information that a reward is granted to the LCS.

the LCS decides on an action and sends a message to the effectors to carry it out.

An LCS is a rule-based learning system and works by maintaining a set of "if-then" rules called *classifiers*. Each classifier proposes an action to take if particular conditions are detected in the environment. The conditional statement of a classifier is a string $\{0, 1, \#\}^L$ where L is the length of the sensor's binary message and # is a wild card that allows the value of the particular bit to be either 0 or 1.

Those classifiers that match the current input message take part in a bidding process (described in Section 2.2) whereby the winning classifier's action is passed to the effectors to be executed in the environment.

The LCS learns correct behaviour using a *reinforcement learning* technique in which the system receives rewards for performing correct actions given the state of the environment. It does this with no prior guidance or knowledge about the problem space, but merely seeks to maximise the reward it can earn. This reward-based technique is adapted from learning techniques often used to train animals or young children. In such situations the reward is granted to the pupil by another individual; similarly rewards are granted to the LCS by an external program, called the *reinforcement program (RP)* (Butz & Wilson 2000).

The LCS uses the reward from the RP to credit the classifier (or classifiers) that proposed the action that brought about the reward. An average reward is recorded for each classifier and it is this value that is used in future bidding processes. In this way, classifiers that propose correct behaviour are more likely to be selected in future.

The system also uses a *Genetic Algorithm (GA)* to perform operations on the population of classifiers to create new, and potentially useful classifiers. As the name suggests, the GA models the genetic operations that occur in the DNA of living organisms. In particular, it creates offspring classifiers that have

a combined version of the characteristics of their parents. During the process there is also a small possibility of a random mutation. These operations are string manipulations on the conditional statements of the parent classifiers.

An LCS can therefore be described in terms of the following components:

- **Performance:** The selection of an appropriate action, given the current state of the environment.
- **Reinforcement:** Rewarding those classifiers that describe behaviour that is desirable/optimal.
- **Discovery:** Striving to discover better classifiers to bring about consistently higher reward or more constant performance.

The problem that an LCS attempts to solve can be categorised as either a single step or multiple step problem. As the name suggests, single step problems only last for a single cycle of input processing, action selection and action execution. After this, any reward earned is awarded and a new problem begins. In a multiple step problem, the LCS may have to go through a number of these cycles before a reward is earned and/or the problem is solved. In both problems types, once a particular scenario has been solved, the problem is restarted a number of times with different starting conditions. During each problem, the classifier population evolves and so should become increasingly skilled at selecting the correct actions. The two scenarios often used as examples, or test environments, for LCS implementations are the single step "Boolean Multiplexer", and the multi-step "Woods" (Wilson 1985) problems. See Appendix C for definitions.

To enable an LCS to support potentially long chains of actions in multi-step problems, Holland proposed a complex message-passing framework in which internal messages could be passed between classifiers as well as the external messages between the LCS and the environment. However, as Wilson (1994) notes, attempts at implementation have met with "mixed success":

> Though a number of researchers were inspired by Holland's framework to investigate classifier systems ... efforts to realize the framework's potential have met with mixed success, primarily due to difficulty understanding the many interactions of the classifier system mechanisms that Holland outlined. The most successful studies tended in fact to simplify and reduce the canonical framework, permitting better understanding of the mechanisms which remained.

Wilson's *Zeroth-level Classifier System (ZCS)* (Wilson 1994) was an attempt to simplify Holland's framework and "to provide a viable foundation for building toward the aims of Holland's full framework". Later Wilson describes the *XCS* (Wilson 1995) form that builds on the ZCS and has been shown to be effective and stable where other LCS implementations have failed. In particular the tendency of an LCS's population of classifiers to be taken over by those that receive high rewards in some situations but low rewards in other areas. These over-general classifiers are selected for reproduction because of the high reward they receive in some areas. However, they do not represent accurate or complete solutions for the problem space. This is analogous to an ecosystem of cooperative populations in the natural world. Populations of organisms will specialise in the various different environmental situations (or *niches*) and live cooperatively

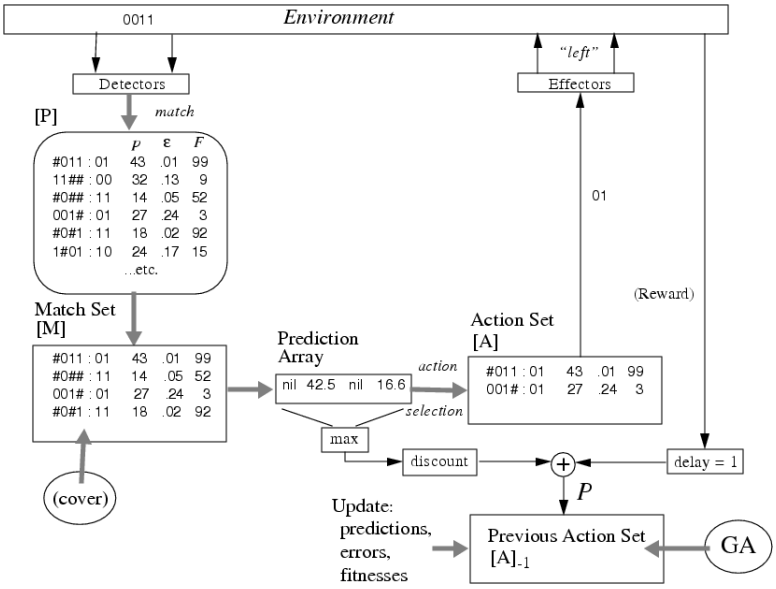

Figure 2.2: Schematic illustration of XCS (taken from "Generalization in the XCS Classifier System", Wilson 1998).

because they are not competing for the same resources. In the same way, Wilson proposed that a classifier system should be able to maintain cooperative sets of classifiers, thus enabling the system as a whole to cover more of the problem space.[1] To enable this to happen, Wilson based the reproductive capabilities of classifiers in an XCS on accuracy of predicting reward rather than the actual amount of reward received. In this way accurate classifiers in all niches have an equal chance of reproducing regardless of the value of the reward they predict.

Figure 2.2 shows the XCS and its components. These are described in detail in the following sections.

2.2 XCS Performance Component

At each discrete time step, the system receives an input message from the environment and forms a subset of classifiers whose condition statements match the current message. This subset is known as the *match set [M]*. For example, the message 101101001, could contain classifiers with the following conditions: 101101001, 10##01001, ####01001, #########.

The members of $[M]$ may propose a range of different actions. A *conflict resolution subsystem* (Barry 2000) is therefore required to select an action to be executed. In XCS this is done by creating a *prediction array* in which is placed the system's prediction for the reward for each action represented in $[M]$. The

[1] Alternatively this can be thought of as a set of rules that provide a complete and accurate mapping of conditions and actions to a reward: $X \times A \Rightarrow P$.

prediction for a particular action is calculated as fitness-weighted average of all classifiers that propose a particular action.[2] Using this array, the XCS can either select the action with the highest prediction or use a random selection process. These two techniques demonstrate the trade-off that the system must manage between maximising reward and evolving an accurate rule-base. By selecting the highest prediction, the system is seeking to gain maximum reward; in a random selection, the system executes sub-optimal rules in order to bring new rules into use and identify inaccurate rules in the population. These different selection techniques are often called *exploit* and *explore*, respectively. In XCS, the parameter P_{exp} is used to denote the probability of using the explore policy in a given cycle.

Following this selection process, the selected action is sent to the effectors to be executed and an *action set [A]* is formed that contains those classifiers from $[M]$ that proposed the executed action. The members of $[A]$ are given a share in any reward earned, and in multi-step problems members of $[A]$ in steps leading up to the earning of a reward are also granted a share. This is discussed in more detail in the next section.

2.3 XCS Reinforcement Component

As discussed earlier, the Discovery Component of an XCS (and to some extent the Performance Component) is based on a classifier's ability to accurately predict the value of reward that will be earned if its action is executed. This fitness value is a function of the relative accuracy of the classifier with respect to other classifiers in action sets to which the classifier has belonged.

In previous LCS theory there was simply a "performance" parameter maintained for each classifier – an average of reward earned – that was used for both action selection and GA selection. In order to calculate fitness in XCS, three parameters are required:

- p: The classifier's reward prediction.
- ϵ: The average error in this prediction (when the prediction is compared to the actual reward that was received).
- F: The resulting fitness value for the classifier.

It is therefore the purpose of the Reinforcement Component to update these parameters for members of $[A]$ after an action has been executed.

In a single step problem, the prediction parameter p is updated for each classifier, j, using the *Widrow-Hoff delta* rule (Widrow & Hoff 1960):

$$p_j \leftarrow p_j + \beta(R - p_j) \tag{2.1}$$

where R is the reward earned and β $(0 < \beta \leq 1)$ is the *learning rate* – a system parameter that, when varied, changes the balance between speed and accuracy of discovery (Butz, Sastry & Goldberg 2003).

However, in multi-step problems, the XCS must reward a chain of action sets that have led to a reward. In order to avoid a complex system involving recording the histories of each classifier, XCS uses the a payoff mechanism from

[2] As mentioned earlier, fitness in an XCS is based on accuracy. See Section 2.3.

the current action set, $[A]$, to the previous one, $[A]_{-1}$. Instead of updating p using a reward value, the payoff value, P, is used. P takes the maximum value from the current prediction array, discounts it by γ ($0 < \gamma \leq 1$) and adds in any reward earned in the previous time step.

ϵ, the error parameter, is similarly updated, using the Widrow-Hoff technique with the absolute difference between the prediction p_j and the reward P (or R):

$$\epsilon_j \leftarrow \epsilon_j + \beta(\mid P - p_j \mid -\epsilon_j) \tag{2.2}$$

Finally, the fitness parameter, F can be calculated:

$$F_j \leftarrow F_j + \beta(k'_j - F_j) \tag{2.3}$$

where k' is the relative accuracy of the classifier within the action set. This is achieved by first calculating the actual accuracy, k, and then dividing k by the number of classifiers in the action set. Accuracy is an inverse function of ϵ and is often calculated as:

$$k_j = \exp((\ln \alpha)(\epsilon_j - \epsilon_0)) \text{ for } \epsilon_j > \epsilon_0 \text{ otherwise } 1 \tag{2.4}$$

where α ($0 < \alpha < 1$) is a system parameter that can be altered to increase the rate of decline in accuracy as error increases, and ϵ_0 acts as a threshold, after which error is considered negligible. The function causes accuracy to decrease exponentially for error values greater than ϵ_0.

However, in the above calculations the Widrow-Hoff technique is only used to update the parameters if the classifier is thought to have had sufficient experience that its parameters have reached a stable value. The threshold value is set to $1/\beta$ time steps, before which the parameters are updated as a simple average of the existing and new values.[3]

2.4 XCS Discovery Component

It is the job of the Discovery Component to introduce improved classifiers into the population – classifiers are more accurate or more general than existing members; and to remove under-performing classifiers – those who are inaccurate or too specific. This process is driven by a genetic algorithm (the GA), which emulates the evolutionary processes found in nature, to produce new classifiers and remove others.

Most reproduction in nature is carried out sexually – in other words, offspring are produced from two parents and inherit a genotype that reflects this combination of the adult genes. The process by which the genotypes of the adults are combined, and so the attributes that the offspring exhibits, is random. However, given parents with some set of useful attributes there is a chance that the offspring will gain a sizable proportion of desirable traits. In this way, the population can evolve better members as the child is likely to be able to out-compete its peers and so propagate its genes to a greater extent.

This principle of "survival of the fittest" is modelled by the GA, by selecting two classifiers from the population who are deemed "fit" and performing a

[3] A consequence of this is that an additional parameter, exp, is required for each classifier. exp is a count of the number of times the classifier has belonged to $[A]$ and is updated during the Reinforcement Component.

crossover operation on their conditional statements. In this process the string is separated into two sub-strings at the same random point on both parents. The second string from each parent is then switched to produce two offspring. During this process in nature, a mistake sometimes occurs whilst combining the genes in offspring. It is possible that this random *mutation* will benefit the offspring and it will bring about a change in the population behaviour in the same way as previously described. Therefore allowances are made for this in the GA: when creating a new classifier there is a small probability that a bit will be changed in the process of replication.

Arguably, it is the way in which classifiers are selected for replication that separates XCS from ZCS and other systems. As we have already seen, classifiers in an XCS have separate fitness and prediction parameters. This allows the GA in an XCS to select classifiers for reproduction based on their relative accuracy and hence eliminate the competitive pressures between classifiers in different niches. In another change from LCS, the GA in an XCS operates on members of $[A]$ rather than $[P]$. This *niche GA* was first suggested by Booker (1989) and used in XCS as a further way to reduce competition between niches. Booker also points out that the offspring of parents in separate niches are less likely to exhibit useful attributes than those with parents in the same niches.

The actual selection of classifiers to be bred is usually done using a roulette-wheel technique to randomly select two parents with a probability proportional to the classifier's fitness. More recently, tournament selection has been proposed as a more effective method of selecting parents. In this process, a number of classifiers are selected at random and the two fittest are then selected from this pool (Butz et al. 2003).

To ensure uniform evolutionary pressure in each niche, the GA is not applied on the formation of each action set. Rather, the average time since the last GA invocation is calculated for the current members of $[A]$, and if it exceeds a certain threshold, Θ_{GA}, the GA is executed.[4]

There are occasions when a classifier is produced that is identical to another classifier in $[P]$. In a measure to save computation time, these classifiers are merged to form a *macroclassifier*. In effect all classifiers in an XCS are macroclassifiers that contain a *numerosity* field that records the number of *microclassifiers* that it represents (initially 1). When a new classifier is generated, $[P]$ is scanned for a duplicate classifier. If found, the numerosity of the existing classifier is incremented and the new classifier is deleted. The terms *classifier* and *macroclassifier* are usually interchangeable. However there are occasions when this distinction is necessary; for example when calculating relative accuracy in the Performance Component, it is necessary to consider the total number of microclassifiers.

An LCS can be seeded with either a population of randomly generated classifiers; a collection of potentially useful ones; or none at all. And in the same way as a natural population has an upper size limit, the LCS imposes a maximum population size; usually referred to as N. Therefore, if a population size is less than N, new classifiers can be generated and simply inserted into the population. However, once the population size reaches N, a number of microclassifiers must first be removed from the population before any new members can be

[4] To facilitate the calculation of this average, each classifier records the time step that the last GA invocation occurred in an action set that it belonged to (referred to as ts).

added. Deletion also occurs using a random-based technique; Wilson describes the use of a roulette-wheel technique in which the probability of selection for deletion is increased if the classifier belongs to action sets whose average size is relatively large and further still if the classifier's fitness is less than a certain small proportion of the population's average fitness.[5]. In targeting these under-performing classifiers, we can expect the removal of inaccurate classifiers and the maintenance of equal size niches. As with subsumption, classifiers are not considered for deletion until their experience reaches a predefined threshold, Θ_{del}.

In a later addition to the XCS, Wilson (1998) describes *subsumption deletion*. This was suggested after the identification of "accurate, but unnecessarily specialised classifiers". This should not be the case as, in theory, general classifiers will occur in more action sets and so out-compete more specific classifiers. However, Wilson suggests that in certain cases, where the inputs received from the environment were sparse, this was not occurring. Wilson suggests that newly created classifiers are checked for specificity in conditions against their parents and the other members of the current action set.[6] If the offspring are simply a less general case of another acccurate classifier they are subsumed – the numerosity of the existing classifier is incremented and the new classifier is deleted.

[5] Again, an extra field, as, is required to record an average of the action set size that a classifier has belonged to.

[6] Wilson defines subsumption of a child by a parent being when the set of possible strings matched by the offspring is a proper subset of the set matched by the parent.

Chapter 3

Existing XCS Implementations

Various XCS implementations have been produced in programming languages such as C (Barry's XCSC and Butz's Illigal C-XCS) and Java (Barry's JXCS and Butz's Illigal Java-XCS).[1] Their uses are twofold: they allow a user to apply their own problem, or test environment, to the XCS algorithm; and they allow a developer to modify the operations of the XCS algorithm for research purposes. This section analyses these implementations from the perspective of these two programmers: the *User* and the *Developer*.[2]

3.1 Java Implementations

One of the main benefits of a Java implementation, over those written in a procedural language such as C, is that it provides an object-oriented code structure to represent data and separate the functionality into logical groups.

Butz uses a minimal structure with regards to the number of object classes defined. The library uses objects to represent a classifier (`XClassifier`); a prediction array (`PredictionArray`); a classifier set (`XClassifierSet`); an XCS (`XCS`); and the collection of parameters `XCSConstants`. Further to this it defines an interface, `Environment`, to be implemented when creating a test environment.

Although many of these classes are similarly defined in Barry's structure, he defines several others in addition. Where Butz has used primitives to represent conditions, actions, messages, and reward; Barry has defined further classes. In doing so, Barry has provided a means for easier development from a Developer's view-point. For example, alternative message encodings can be researched and developed by simply modifying the `Message` class. Providing the object implements the same methods, this data cohesion means that the programmer should not have to worry about the effects of the change in any other part of the system. Barry also includes a more flexible reporting mechanism to allow for any

[1] See Appendix A for details

[2] In this section, and for the remainder of the project, the words "User" and "Developer" have been capitalised so as to distiguish them from their more general usage, when used in reference to the two user groups identified.

output format.[3] However, a significant draw-back in Barry's library is that it only handles single-step problems. Although there is probably no reason why this could not be added at some point in the future.

A significant difference that a User will encounter when creating test environments in both libraries is that Barry does not use an interface to define how an environment class should be written. Instead, test environments are subclasses of an `Environment` class. In terms of object-oriented design, Butz's solution does seem to solve the problem in a neater way; allowing the User to create a new class however they like, providing they implement the vital methods to communicate with the XCS. Barry's solution does have the advantage of being able to lay-down some base code and variables that allow the User to gradually create a class that overrides the superclass, instead of having to implement a class in one go.

An interesting difference in approach is also found in the way they deal with sets of classifiers. Butz uses a single class to represent all sets; $[P]$, $[M]$ and $[A]$. Three separate constructors handle creating $[P]$ as an empty population with a specified number of actions; $[M]$ by specifying (amongst others) the current input message; and $[A]$ by specifying the selected action. Each set also keeps a reference to its parent set. Conversely, Barry uses separate classes to define $[P]$, $[M]$[4] and $[A]$. Again, Butz's implementation has, arguably, the purer object-oriented approach; there is presumably some redundant code when defining two classes to handle classifier sets. However it does mean that the class only contains functionality specific to the particular type of set, whereas with Butz's `XClassifierSet`, it was necessary to include functionality relating to $[P]$, $[M]$ and $[A]$.

Both implementations provide some standard test environments. Both provide the single-step Boolean Multiplexer, and Butz includes the multi-step Maze environment.[5]

3.2 C Implementations

Both implementations use data structures to represent similar objects to their respective Java implementations. However, the readability of the code suffers from the syntactic obscurities involved with C programming and are a long way from the algorithmic description described by Butz & Wilson (2000). This means from the point of view of both the User and the Developer—particularly the Developer—significant time would need to be devoted into acquiring implementation-specific knowledge and becoming familiar with the code.

The benefit of using C, however, is its speed. The fact that C is a relatively low-level language does not help readability, but it does mean that we can expect a faster execution time from the resulting program. With an XCS that iterates through thousands of cycles per experiment this is a significant factor, especially

[3] He also defines a separate user interface package that reports information to a user graphically. This is a useful feature, but for the purposes of this comparison, we will focus on the XCS package.

[4] Although the Match Set is combined with the Prediction Array in the `SystemPrediction` class.

[5] This includes a useful text file parsing functionality to allow a User to create new mazes easily.

if the programmer expects to need to run an experiment many times in order to collect thorough results.

Butz attempts to split the functionality within his code into functionality for classifier lists, action selection, environments, XCS constants and the general XCS system. This does make Butz's code slightly more modularised, as Barry keeps all the functionality in a single file. However, like the Java implementation, Barry's libraries are designed more flexibly; with tighter cohesion allowing a programmer to safely modify a component without fear of interfering with the results of others.

3.3 Conclusion

An ideal XCS implementation would:

- **Minimise syntactic clutter** of the implementing language and represent as closely and as clearly as possible the algorithmic processes in the system.
- **Employ flexible object-oriented techniques**, specifically tight cohesion of data that aids development within modules; and low coupling of objects for flexible use.
- **Fast execution of code** at run-time.

These attributes of a good implementation are often contradictory. For example in producing code that is readable, it may be necessary to use extra statements to clarify the semantics of a code block but which may also increase its execution time. It is therefore a matter of finding a suitable balance between the three attributes. We have seen that the existing implementations are perhaps lacking with regards to the first point; that of representing the process clearly and algorithmically.

Part II

Design

Chapter 4

Requirements Analysis & Specification

In Chapter 2 the two stakeholders in the framework were identified as the User and the Developer. The former being interested in setting up environments for the system to act in, and the latter wishing to adapt the behaviour of the system for research purposes. In this section these activities will be further discussed and analysed to form the basis of the Requirements Specifiication on which the framework will be developed. The Specification can be found in Appendix B (p. 81).

Before exploring the specific requirements of these two users, it is worth discussing some of the aspects that are common to both groups. Including the positive attributes discussed in the previous section.

4.1 General Requirements

Framework Code

The framework is an XCS framework and so obviously its primary requirement is that it should contain a working XCS algorithm as a basis for any development. Implicit as this may seem, it is worth remembering that XCS is a branch of the LCS tree and there are also several variations in its implementation. The framework will be based on the papers by Wilson (1998) and particularly the algorithmic description proposed by Butz & Wilson (2000). Any deviations or enhancements made to the core functionality should be clearly explained and justified. As described in Chapter 2, one of the core development objectives for the framework is to maintain this algorithmic description and readability of code. However, every effort should be made to keep the code as efficient as possible. This is especially true of the core XCS algorithm which is executed many thousands of times.

Another general requirement that should be placed on the framework is for it to be written in an object-oriented way. The paradigm is so commonly used in modern programming that it needs little justification here, except to say that failure to use it is very likely to inhibit the framework's use and further development.

Documentation

Documentation is also vital to the framework's usability as an XCS development tool. With many programming languages, automatic documenting tools are available to convert structured code comments into either hypertext or static documentation. By including substantial commenting on all functions within the framework, the need for single line comments within the function body is reduced. Again this is important for the core XCS algorithm where the readability of comments may compromise the readability of the algorithm.

4.2 "User" Requirements

The User should only be concerned with creating an *external* environment for the XCS to operate in. The framework should therefore be encapsulated in such a way that the User should then able to "plug in" a newly created environment and expect the correct XCS behaviour to ensue. In order for this to happen, the framework must define an interface from which the User can develop their environment. From the XCS algorithm described in the previous section, we can see that this interface consists of the following:

- **Supplying a percept** of the current environment state.
- **Performing the action** proposed by the system.
- **Rewarding the system** appropriately for its advice.
- **Informing the system** that a problem has ended.

As there are some system parameters that are problem-dependant, the environment must also expose the number of actions it is able to perform, the length of a percept message, and the maximum reward that is available.[1]

However the XCS can only be expected to successfully solve the problem if the reward schedule created is providing the correct reinforcement to the system and the system parameters are set correctly. This has two consequences for the framework. Firstly the system parameters should be easy to modify. Although it is unlikely that they would be modified during an experiment, it is possible that modifications would be required between experiments without the need to revisit the code. Secondly, as the User is often concerned with analysing the performance of the system and the rules it has evolved, the framework must also have a good reporting system.

There are two common types of report that a User may require from an XCS experiment and that the framework should include: the final population of classifiers; and some step-by-step performance statistics. These performance statistics are commonly the system's performance (either in terms of reward earned or, in multi-step problems, number of steps taken), the system's error (the difference between the predicted and actual reward), and the macroclassifier population. The User should also be able to write their own statistical functions, using data from the classifier system, and register them to occur on particular events during the experiment.

[1] This is used when calculating the value of ϵ_0.

4.3 "Developer" Requirements

In order to gather requirements from a Developer's perspective, current areas of XCS research should be considered in order to identify the type of modifications that a Developer is likely to require, and so develop a framework in which they can be made relatively easily. Three such topics that I have considered in particular are continuously-valued inputs (Wilson 1999), messy coding (Lanzi 1999) and tournament selection (Butz et al. 2003).

XCSR – Continuously-Valued Inputs

Wilson (1999) discusses the development of a classifier system, XCSR, that uses continuously-valued inputs, as opposed to the binary values used in a traditional system. Such as system would be much better suited to real-world scenarios where a sensor is not simply "off" or "on", but have a quantitative value. As Wilson speculates:

> Continuous variables such as temperature, concentration, or age may be decisive in classification, with certain ranges of the variables implying one class and other ranges implying another.

In his paper Wilson suggests using "interval predicates" in place of each bit in a traditional XCS input string. Each predicate, $int_i = (c_i, s_i)$ where c_i and s_i are reals, specifies a rule such that a classifier matches an input if and only if $c_i - s_i \leq x_i < c_i + s_i$ for all x_i. c_i is thought of as the centre value of the interval, and s_i as the "spread" or delta defined with respect to c_i.

The mechanics of an XCSR implementation only differ to that of XCS with respect to its condition matching, mutation operator and covering mechanism. Condition matching is simply a matter of implementing the logic described above. Mutation involves adding a random small amount to each allele with the usual probability. Covering occurs in at the usual times, generating a value of c equal to the current situation and s as a random number typically between 0 and 0.5.

XCSm – Messy Coding

Lanzi (1999) proposes a classifier system in which the chromosomes of the classifiers are not of a set length. In such a system, a chromosome may not specify every gene (under-specification) and may specify some genes more than once (over-specification). In doing so, the relationship between the position of bits in the classifier's condition and the position of sensor bits is removed. Lanzi experiments with knowledge reuse as one of the benefits that this brings:

> Accordingly, classifiers evolved to solve a certain problem can be reused in another application assuming that the tags of the messy genes are still valid in the new application.

He goes on to experiment with a maze in which only due north, east, south and west are available actions. After the population fail to find an optimal solution, the four other movements (NE, SE, SW, NW) are made available to resulting

population. He showed that using only mutation to introduce the new actions, the population learned an optimal solution.

Thus the primary difference between XCS and XCSm is that tags are required in the messy coded chromosome to associate it with the sensor as its position can no longer be used. Each gene now consists of a pair; the first element denoting the gene number (or sensor's identification) and the second giving the value of the allele.

Matching is also embellished to deal with over- and under-specified genes. Genes that are under-specified are treated as though their bits were "don't care" symbols. If a gene is over-specified, the classifier is only matched if the current situation matches all of the alleles.

Covering is handled in the same way buy uses a probability, P_s, that each gene is present in the chromosome at all. Lanzi uses a relatively high value for P_s, of 0.8 or more.

The crossover mechanism in XCSm works in much the same way but uses a modified probability of being invoked. In XCS the probability of crossover occurring is a fixed probability, χ, whereas XCSm uses a calculation: $p_k(\lambda - 1)$ where p_k is the probability of cutting a single gene and λ is the length of the shorter chromosome. Mutation is modified so that the tag can also be mutated with probability μ, and genes can be added and removed from the chromosome, also with probability μ.

XCSTS – Tournament Selection

Tournament selection in XCS (Butz et al. 2003) is an alternative selection mechanism to the fitness-weighted "Roulette Wheel" that is traditionally used to select parents in the GA. To select a parent using tournament selection, a proportion of the population is randomly selected and the fittest member of this subset is chosen. It is argued by Butz et al. that tournament selection makes the process more independent from the problem and system parameter values.

This modification introduces a system parameter, $\tau \in (0, 1]$, that specifies the size of the tournament as a proportion of the total population. It modifies only the selection mechanism, no other aspects of the system are affected.

Returning to the Requirements

Of these modifications the first two—real-valued inputs and messy coding—exemplify modifications at the classifier level; they make no difference to the higher level XCS theory. This highlights the need for a neatly encapsulated model that makes modification via inheritance simple and logical. All functionality that directly manipulates the classifier values, especially the condition, should be enclosed in the same place.

Conversely, tournament selection is an example of a modification that affects a higher level XCS process. Modifications to mechanisms such as parent selection, the GA's macro-operations and action selection should be easily changeable. This is likely to involve "pluggable" functions to allow for run-time alterations. Although modification through inheritance could equally be used.

All three of these research areas also introduce new system parameters. For example, Wilson uses m in the mutation operator for real-valued classifiers as the maximum change possible for each interval, and in Butz's tournament selection

τ is the tournament size. Therefore the input system for system parameters in the framework should be easily extensible and allow additional parameters to be added.

Chapter 5

About Python

Python is a high-level, interpreted and object-orientated language, developed in 1990 by CWI, Amsterdam. Its current development is very much open-source and is owned by the Python Software Foundation (PSF). In Python's introductory documentation[1], the PSF claim that:

> [Python] has an elegant (but not over-simplified) syntax; a small number of powerful high-level data types are built in. Python can be extended in a systematic fashion by adding new modules implemented in a compiled language [...]

This section is an evaluation of the Python programming language with respect to the attributes identified in Chapter 3 in order to justify it as a good candidate for an effective framework.

5.1 Code Aesthetics

It is a reasonably safe assumption that users of the framework would have a good grasp of programming concepts, as XCS is itself deeply routed in Computer Science. However it cannot be assumed that they have any experience of the implementation language. Python is well-known for the ease at which it can be learnt; it has a relatively compact syntax with few exceptions, shortcuts or anomalies. It therefore makes it a good candidate for the framework as code should be understandable, at least at a high-level, by an experienced programmer who has at least a basic knowledge of Python.[2]

Python has several features that differentiate it from most high-level languages around today. It uses indentation to form blocks of code rather than explicitly enclosing them in brackets as is the case for most C-like languages. Similarly, expressions are separated using new lines rather than using a delimiter such as a semi-colon. This has a big impact on the readability of Python code. Not only does it remove the need to use these delimiting symbols in the code,

[1] http://www.python.org/doc/Introduction.html

[2] If a "basic knowledge" definition were needed a suitable bench mark would be the Python Software Foundation's tutorial (http://www.python.org/doc/2.4.1/tut/tut.html). A reasonable requirement for the user would be an understanding of Chapters 3-5 for the basic syntax and Chapter 9 for its object-oriented principles.

it also enforces the practice of indentation that is simply suggested in other languages. However this implicit notation should also be treated with caution as it also means that any mis-alignment of statements will cause the code to behave in an unexpected way. For example, statements can "fall off" the end of a conditional or looping block.

As with many interpreted languages, Python is dynamically typed and variable declarations are implicit. Although an explicit declaration provides a guide to the reader as to how to expect the variable to be used (e.g. numeric data versus a string), it does take another step to removing syntax that is not directly related to the algorithm being implemented.

The language has several useful features built into it that help make programs less verbose. It supports optional function arguments with default values if no actual argument is supplied; automatic code documentation using a string placed in the first line of a function definition; high-level data types such as lists, tuples, sets and dictionaries; and concise list comprehension, making iterating through lists simple and concise.

5.2 Object-Oriented Support

When programming in Python, the developer has a choice whether or not to use the object oriented paradigm. Code can be written procedurally and can be kept manageable in a sophisticated system of modules and packages. However, as we have already identified, it is often preferable to encapsulate data and functionality into objects. Python's object oriented support is comparable to many other such languages, such as Java, but in a less strict way. The official tutorial[3] states:

> [C]lasses in Python do not put an absolute barrier between definition and user, but rather rely on the politeness of the user not to "break into the definition."

As an example of this, Python has no explicit concept of private or public data members within objects. However, by inserting two underscore characters before a variable, the compiler prefixes the variable name with the class name. This allows a class to define private variables but does not prevent an external process access to the variable by prefixing the class name manually.

In other ways however, Python has object oriented support at a much deeper level. All data types, including primitives, are objects and can be subclassed. This includes classes themselves; enabling classes to be passed to, and return from, methods. Python also supports multiple inheritance; allowing one class to inherit data and functionality from more than one parent class.

Python does not, however, support Java-style interfaces that define rules for the methods an object should contain. Such an interface was used in Butz's Java library to define how an object that represents an Environment should be composed. However, it is possible to emulate the behaviour of an interface using Python's multiple inheritance with abstract classes that contain "empty" functions. This does not, however, force the programmer to implement each method as a true interface would, but provides some guidance to the programmer.

[3] http://www.python.org/doc/current/tut/node11.html

In summary, we can see that the language provides a level of support that would be conducive to producing a classifier system in the object oriented paradigm. It is also possible to produce manageable and readable procedural code if areas are unsuitable for an object oriented approach.

5.3 Execution Speed

Python is an interpreted (or scripted) language and, similarly to Java, it produces a platform-independent byte-code that is executed by a virtual machine.[4] This means that Python programs can be run on any platform that has an interpreter installed. However, this flexibility means that the execution speed of Python is slow in comparison to languages such as Java and C. There are several ways to address this problem:

Code Optimisation

As with all languages, optimisation in the use of code is a technique used to increase performance. One of Python's standard library items is the `timeit` module. It contains functionality to accurately time the execution of pieces of code. Using this, differing techniques can be analysed for speed and efficiency.

There are several sources for Python code optimisation available. Most centre around efficient methods for using loops and for string and list manipulation. Lundh (2005) and van Rossum (2005) propose a variety of optimisation techniques. For example, Python is written for fast list processing so it is often expedient to handle data in lists rather than string or other representations. Also, the `map` function can be used to apply a function to all items in a list. This is often more efficient than using a control loop as the `map` function is written in C. However, a trade-off exists between the speed gain and the readability of code.

Python Compilers

Python "Just In Time" compilers such as Psyco[5] have been developed that generates machine code at run-time rather than interpreting the Python code. Psyco is designed to be as transparent as possible and its developers boast anything from double to a hundred times the performance of Python to that approaching C. However, to take full advantage of its increased execution speed requires some changes to the source code that could, in theory, impact on the readability of the code.

Modules in C

As with many high-level scripting languages, Python has the ability to use components compiled in other languages. For the purposes of an XCS implementation, stable modules – that are not being developed by the programmer

[4] Although the byte-code is generated at run-time rather than pre-compiled as with Java

[5] See http://psyco.sourceforge.net for more details

– could be written and compiled in a language such as C. For example if a Developer was seeking to improve certain processes in the GA, they could use C modules for the rest of the XCS.

5.4 Conclusion

Having addressed these issues, we can see that Python is a good candidate for an XCS implementation. However, it is not the only language of its type; many of its advantages are exhibited by languages such as Lua, Perl, Ruby and Tcl. These last two languages in particular would also be suitable candidates for an implementation. There is a good deal of debate between supporters of all four languages along similar issues of the readability of code, purity of object orientation support and execution speed. It is therefore difficult to conclude with certainty that Python is the better of the three, however it does have a couple of advantages over Ruby and Tcl with regards to an XCS implementation.

- Ruby is a not as well established as Python. It was first developed in 1995 in Japan, but has only become popular since around 2000. In terms of a useful XCS library, it would seem wise to select a language with a more established support and developer community.
- Ruby is a pure object oriented language. Although Python has deep object oriented concepts, Ruby is a more pure approach in the sense that there are no exceptions to the rules. For example the number 1 is an object of type `Fixnum`. It is likely that these strict rules will lead to code that contains unnecessarily convoluted (and unintuitive) code in some cases.
- Tcl was first developed for use as an embedded language and has been adapted for use as a language in its own right. It therefore as a very simplistic syntax making it easy to write small programs but can become difficult for larger projects.
- Tcl does not have an intermediate byte-code. The code is therefore interpreted each time it is run. Whereas Python stores the byte-code for a faster execution on successive runs.

Although these are not reason in themselves for dismissing implementations in such languages, it perhaps provides sufficient reason for choosing Python as the language in which to implement an XCS library in this project.

Chapter 6

Top-level Design

This section describes, at a high level, an object-oriented design for the framework and provides justification for these design decisions.

The main components of an XCS experiment were described by Butz & Wilson (2000), as detailed in Chapter 2 (see also Figure 2.1 on page 8). Between them, these high-level components contain the environmental data and mutators, reinforcement schedule and XCS algorithm required for an XCS experiment:

- An **X-Classifier System**: The action-selecting agent containing the performance, reinforcement and discovery components.
- The **Environment**: Provide an interface between the classifier system and the "real world".
- The **Reinforcement Program**: Provides feedback in the form of reward to the classifier system for actions it has done.
- A set of **Sensors**: The mechanisms that produce the environment percepts (or sensor readings).
- A set of **Actuators**: The mechanisms that act on the environment.

These components make ideal concepts with which to base the main classes for the framework. The following section introduces and justifies these proposed classes and their interactions.

6.1 The Environment

The Environment should provide all information about the external environment (or the "real world"). In the concept described by Butz & Wilson (2000) the sensors and actuators that actually interact with the environment are part of the classifier system. However, in this framework they will be part of this Environment class. This was done to separate the XCS from the environment as much as possible. In theory, to have been conceptually correct the classifier system could have been designed to be instantiated with a list of function objects, indexed numerically, that it would fire at the appropriate time. This, however, seems to be overly elaborate just to maintain a more conceptual model. It would also have made a significant breakaway from the existing XCS implementations

and could have formed a barrier to the framework's use. Therefore the sensor and actuator functionality is encapsulated in this Environment class.

The class' interface will consist primarily of two public methods: one to execute an action; the other to generate and return the current state of the environment. In addition it will also need to expose some properties to provide the classifier system with some initial information. These are the number of actions that it is possible to execute (n) and the length of a percept (L).

Because the environment and reinforcement program concepts are specific to the particular environment they represent, this Environment class and the Reinforcement Program class that follows are abstract. This means that they would need to be made concrete (i.e. overridden with actual functionality) by the user when implementing a specific environment into which the XCS will be applied.

6.2 The Reinforcement Program

The Reinforcement Program class consists mainly of two definitions that provide the classifier system with feedback on its performance. The first is a function that returns a reward based on a proposed action and the current environment state. The second element of the class is a public property that informs the classifier system whether the current problem has been solved. In single-step problems this will always return `True`. In multi-step problems this will depend on whether specific conditions have been met to end the problem.

Specifying a separate class for the reinforcement program is another trade-off that had to be made between keeping the framework conceptually similar to its written description and not confusing the user by diverging from the norms of other XCS implementations. In this case it was decided to make a distinction between the two classes knowing that, owing to Python's multiple inheritance mechanism, an aggregate class can be formed that would be equivalent to the `Environment` classes defined in the implementations of Butz and Barry.

6.3 The X-Classifier System

The X-Classifier System's main operation is to run through one cycle of the XCS algorithm – action selection (performance), reinforcement and discovery – as described in previous sections.

It differs conceptually from Butz & Wilson (2000)'s description in that its main "run" method in fact only runs through one cycle – reads a percept, suggests an action and updates its rules depending on the result. The amount of times it does this has been put into the control of a higher controlling object. The reason for this alteration was to extract the less defined "termination criteria" from the classifier system and reduce the likelihood that a user will need to customise this component. Instead termination of the experiment is controlled by a higher level component (such as the Experiment class described in the next section) that repeatedly asks the system to select an action.

6.4 The Experiment

In addition to these classes that were based on Butz's description, a controlling object is required that initialises each of these components, starts and terminates the experiment after a number of problems/steps and facilitates reporting the results. The class is not designed to be subclassed, but allows most of its settings to be modified at run-time. To run an experiment at least some of the following features are required:

- An XCS object
- An environment object and a reinforcement program
- Termination criteria
- System parameters
- Step, episode and experiment listeners

Of these features, only the environment and reinforcement program need to be supplied for an experiment to run. The XCS object can also optionally be supplied if the user has a modified algorithm to use. The termination criteria is set to a default that ends the experiment after a number of steps. The system parameters can be optionally specified to override the default values. In addition, the Experiment class should allow users to register listener functions to be fired either at the end of every step, every episode (on multi-step problems) or at the end of each experiment.

The experiment also creates two reports that are generated, if requrested by the user: the final classifier population and the performance statistics. These are written to a file specified by the user.

Part III

Implementation

Chapter 7

Environment & Reinforcement Program

As a first step in implementing the framework, we decided to develop the environment structure on which the XCS component would be based – the `Environment` and `Reinforcement Program`.

7.1 The Abstract Classes

The interface for the abstract `Environment` and `ReinforcementProgram` classes are shown in Figure 7.1. It also shows how these classes can be used to form an aggregate data type that can be used as both the environment and reinforcement program.

The Python language does not provide explicit syntax for creating abstract classes, as is the case in other languages such as Java. Instead the classes will simply be treated as though they are abstract and, in-keeping the Python ethos, we will leave it to the politeness of the user not to "break into the definition". There is an exception type provided in Python for this use; the `NotImplementedError` is placed in the function definitions of the abstract class. If the user attempts to use any of the functions from the abstract class, this exception will be raised.

In addition to the `EOP` property, it was also decided that other information would need to be exposed in order to produce statistics during the experiment. Therefore the maximum number of steps expected, and the maximum and minimum rewards possible were added as properties to the `ReinforcementProgram` class.

Another addition was a `reset` method to the `Environment` class. It was decided to add this explicit reset mechanism mainly for multi-step problems in order to allow the user to define exactly when the environment starts the next problem. Although this is not as necessary in single step problems, where the end of a problem occurs on each step, it was thought that implicit resetting caused confusion in multi-step problems as it is not obvious when the reset should occur.

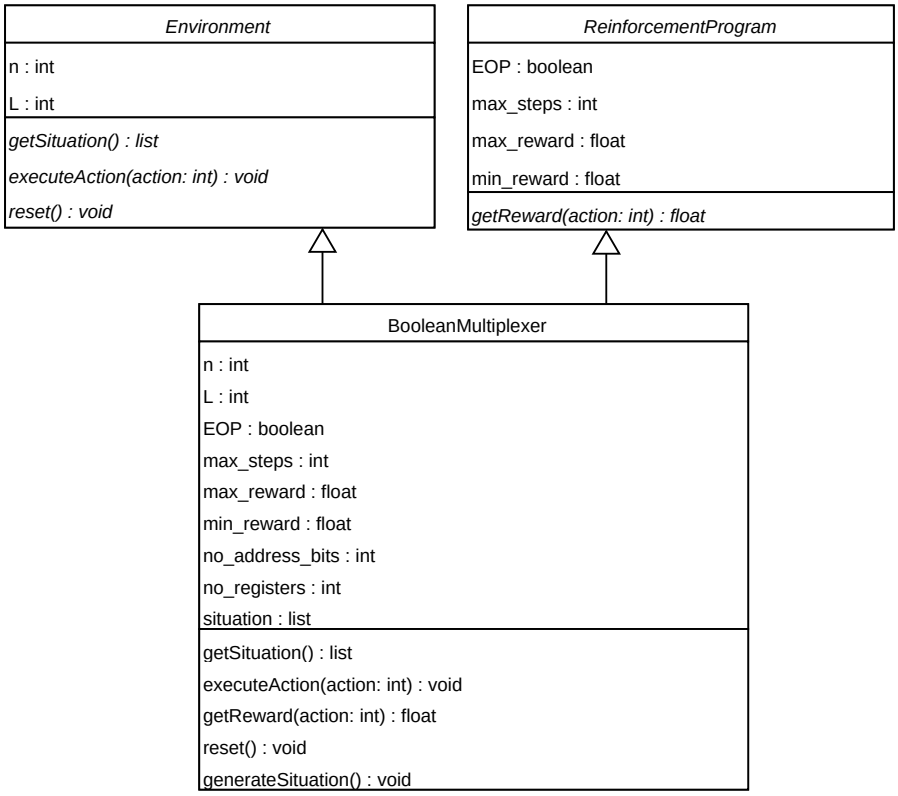

Figure 7.1: The `BooleanMultiplexer` class inherits from the `Environment` and `ReinforcementProgram` classes to form an aggregate type.

7.2 Situation and Classifier Condition Format

A fundamental design decision needed to be made at this point; the format of the message (σ) that is passed to the classifier system. Traditionally this is a binary string composed of the sensor readings. However, as described in Section 4.3 (page 21) there is currently a good deal of research into alternative encodings for the percepts and the closely related classifier conditions.

An alternative to a string would be to use Python's list type to represent percepts and conditions. This has two main advantages over a string: lists in Python are mutable, strings are not[1]; lists elements can be of any data type, including other lists or tuples.

The immutability of strings mean that we would expect the cut and splicing operations that take place in the GA to be more memory intensive than if they were performed on a mutable data type such as a list. Mutability would also mean that the code generated to manipulate the conditions should be less convoluted. During the crossover process, for example, the two conditions can be switched directly allele-by-allele, rather than having to construct two new conditions and reassign the classifier's condition to them. However, the efficiency savings of lists during the GA are also likely to cost time during other operations such as iterations, counting and copying where the more complex data type has a disadvantage.

The decision to use lists over strings was made primarily owing to the greater advantages of having an extensible format for encoding and the potential for clearer code, over the more marginal efficiency issues.[2]

7.3 Environment Types

Classifier system research generally centres around two main types of problem: the single step "Boolean Multiplexer"; and the multi-step "Woods" problems. For a detailed description of these problems see Appendix C. These are therefore the two environments that need to be included as standard in the framework and that will be used later on in the project for testing purposes (See Chapter 12).

As Figure 7.1 shows, the `BooleanMultiplexer` is inherited from the two environmental abstract classes. It accordingly redefines the abstract methods and properties with functionality and data for the Multiplexer problem. The class has been designed to be as flexible as possible and can vary in size (6, 11, 20, etc) by supplying a value of k to the constructor. A random generator is used to create situations when required and these are tested against an action when `getReward` is called. A reward of 1000 is returned for the correct action and 0 otherwise.[3] The `max_reward` and `min_reward` are therefore set to return

[1] In other words, strings cannot be modified once they are created. Any mutating operations, such as concatenation) applied to the string result in the creation of a new string

[2] This decision was also helped by a discussion on the `comp.lang.python` newsgroup and having taken into account much of the literature on the problems of string manipulation in Python. The interested reader may like to read: `http://www.python.org/moin/PythonSpeed/PerformanceTips`.

[3] There are variations on this reward schedule; we have implemented the most common. To modify the environment the user would simply have to further subclass the `BooleanMultiplexer` and override the rewards.

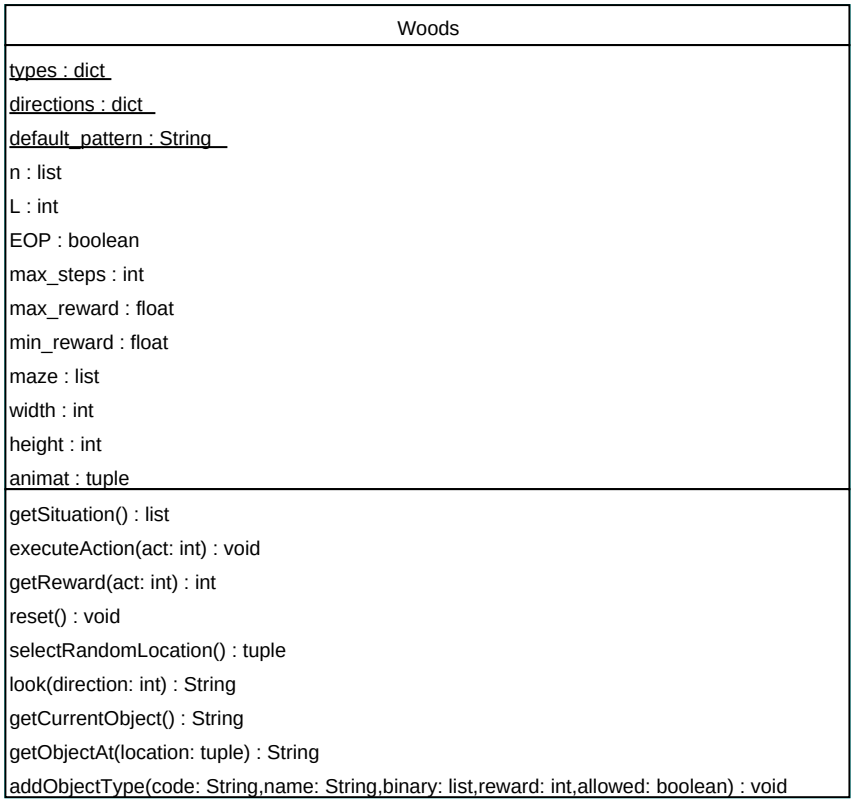

Figure 7.2: The `Woods` class.

1000 and 0 respectively, and the `EOP` property always returns `True`.

Figure 7.2 is the `Woods` class, which is a more complex derivation of the `Environment` and `ReinforcementProgram` classes. An object is instantiated using a text file that specifies the layout of the maze to use. The `types` property is a dictionary of object types that can appear in the maze and by default contains the food and rocks of Woods2 (although more can be added using the `addType` method. The `animat` tuple is used to keep track of the creature's horizontal and vertical position within the maze and when `executeAction` is called its co-ordinates are modified accordingly. This movement has a few subtleties. Firstly the maze pattern repeats indefinitely, so when the animat's co-ordinates must loop around the pattern in both directions. Secondly, the movement must be checked against the type of object on the square to see if it is a legal move (i.e. not a rock). Whether or not the animat can move onto an object type is specified in the `types` dictionary. The `types` dictionary also contains the reward and binary encoding for each object type that is used when the `getReward` method is called and when building the percept list, respectively.

Both environments have test applications written alongside their definitions. If the modules in which the respective classes are defined are run (as opposed to being imported), an interactive test environment is started.[4] This allows the

[4] This follows a popular Python convention of including test code when a module is run

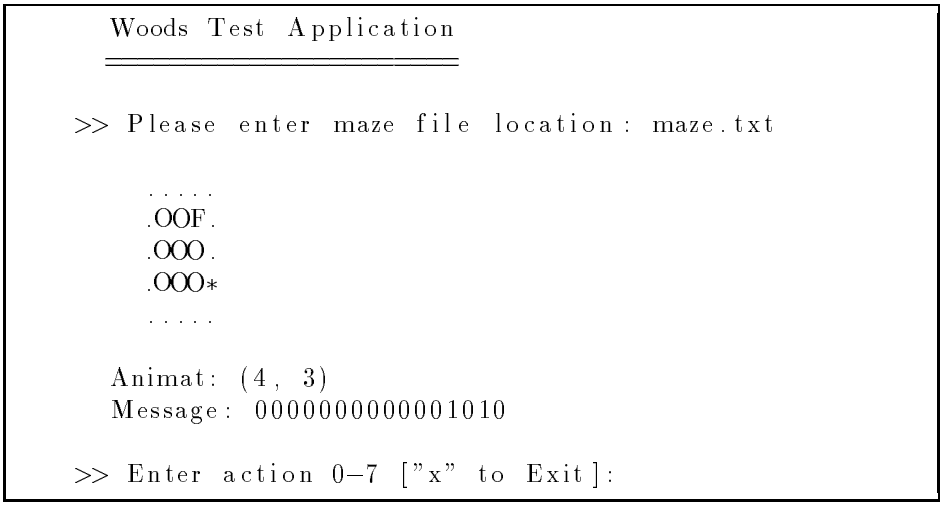

Figure 7.3: Output from the `Woods` test application.

user to control the environment in the same way as the XCS will be able to. The user is simply fed a continuous stream of problems to solve and is prompted for an action and informed of the reward due for the selected action.

instead of imported. We have tried to apply to this framework wherever possible.

Chapter 8

Procedural XCS

In order to produce an XCS implementation that was an accurate representation of the algorithm described by Butz & Wilson (2000), a preliminary procedural version was produced in Python.

8.1 Design

This module, named `ProceduralXCS`, contains the suite of functions described in the aforementioned paper. For its environmental information it uses the `Environment` and `ReinforcementProgram` classes defined in the previous chapter. It also defines a `Classifier` class to encapsulate data relating to an individual classifier as defined by Butz: condition C, action A, prediction p, error i, fitness f, experience exp, timestep of last GA ts, average action set size as and numerosity num. Also, for completeness it also contains k, the accuracy value.

This version is, where possible, a direct translation of the described algorithm into Python. However, some adaptions were necessary in order to produce this working implementation. The remainder of this section documents and justifies these adaptions.

Main Loop

As described in Section 6.3, the largest alteration made to the algorithm in this implementation was the fact that control of the main loop has been moved to a user-defined function. In practical terms, the RUN EXPERIMENT procedure from the paper is implemented as the `run` method which contains the equivalent of lines 2-21 of the algorithm (the body of the main loop). For the suite to be used, the user should write their own loop – with their own termination criteria – and call the `run` method within it. Although this creates more work for the user, it allows the module to be more easily customised with various termination criteria.[1]

Also added to the main `run` method is a step count that is set to force the end of a multi-step problem if it exceeds the maximum step count specified by the environment. This mechanism simply prevents the system becoming stuck

[1] In Chapter 10 (p. 53) a wrapper class is created that provides the functionality for this outer loop.

in a loop during the initial phases of an experiment because it has not built up enough rules.

Action Selection Strategies

Butz & Wilson (2000) differentiate between exploration and exploitation cycles when choosing an action, the cycles are not treated differently during the reinforcement and discovery phases. This is the case in both Butz and Barry's implementations: the GA is only run on exploration cycles, and reinforcement is also only run on exploration cycles in single-step problems. In addition, it is not usual for the action selection strategy to change between steps in a multi-step problem as is the case in the described algorithm.

It was therefore necessary to add these clauses into the main loop. Also, the choice of strategy has been removed from the `selectAction` function and moved to the start of the cycle where it checks whether a new selection is required (i.e. whether it is the start of a new problem).

Action Set Subsumption

A check for subsumption of a classifier by members of the Action Set is carried out on each cycle in the algorithm. However, Butz and Wilson also comment that:

> Besides checking if an offspring is subsumed by a parent, one could also check if it is subsumed by other classifiers in the action set, or even the population as a whole.

This was Wilson's original proposal for subsumption and this is a technique used in other implementations such as Barry's JXCS. Therefore this check has been added to the GA algorithm in the form of the `actionSetSubsumeInsert` function which does this check before inserting a new classifier into the Population. As this is not a standard part of the algorithm, a boolean parameter, `do_GA_subsume_insert`, has been added to control whether it is used or not.

Classifier Condition Format

As discussed in Section 7.2 (page 34) the percept (σ) received from the environment is encoded in a list of 1's and 0's (or `True` and `False` values which have the same numerical value) indexed from left-to-right. In order to encode a classifier's condition, a wild card convention is required. For this it was logical to use Python's `None` data type. For example:

01#1## → [False, True, None, True, None, None]
11###0 → [True, True, None, None, None, False]
→ [None, None, None, None, None, None]

Global Variables

In order to facilitate the use of these procedures the following global variables are defined in the module and initialised in a `setup` function:

```
    mu = _params['mu']
    n = _params['n']

    i = 0
    while i < len(cl.C):
        if random.random() < mu:
...
```

Figure 8.1: The values of the learning parameters are stored into local variables of the same name before being used in the main algorithm. This simplifies the reading of the code.

- `_env`: The environment object.
- `_rp`: The reinforcement program object.
- `_t`: The current timestep.
- `_params`: A dictionary containing the learning parameters required to run the experiment.
- `_record`: A dictionary that records the data during a cycle of the main loop.

The first four variables are present to give access to vital information from any method. It is done in this way, rather than passing the values into the sub-procedures as arguments, to minimise the code and to keep it close to the algorithmic description which also gives these variables global scope. As described later, the final variable is used as a means of keeping state between function calls.

In order not to interfere with the algorithm within a procedure, all retrieval of values from a dictionary is done in the first line (or lines) of code and stored in a local variable (as shown in Figure 8.1). This means that the local variable can be used in the body of the algorithm in the same way as was described in the paper, rather than the less obvious code involved in retrieving the value from the dictionary.

This convention was extended to procedures in which preliminary calculations were required before the body of the algorithm could be run. For example, on line 8 of the GENERATE COVERING CLASSIFIER procedure: $A_{cl} \leftarrow$ `random action not present in` $[M]$. In such cases it is difficult—and often inefficient—to include the semantics in-line when implementing the statement in a programming language. Where possible in the implementation, the value of these statements has been moved to the beginning of a procedure. The following code segment, taken from the UPDATE SET procedure, is an example of this. It would be inefficient, as well as distracting to the reader, to calculate the total numerosity in its described location as it is repeated unnecessarily for every classifier in the set.

`for each classifier` cl `in` [A]

...

$as_{cl} \leftarrow as_{cl} + \beta * (\sum_{c \in [A]} num_c - as_{cl})/exp_{cl}$

Another design decision regarding the system parameters was the naming convention to use for the Greek symbols of the parameters. In many implementations the name of the letter is simply written in English. However this is not very succinct and does not lend itself to symbols such as θ_{del}. Therefore, where possible the letter is used in its Latin form (e.g. "a" for α, B for β, etc) and the LaTeXconvention of using an underscore for subscript is adopted (e.g. "T_GA" for θ_{GA}). Where this is not obvious the name of the symbol has been used (e.g. "nu" for ν).

In the algorithmic description, the population, action set, reward and situation are stored on each iteration of main loop for (possible) use on the next step. Because we have moved control of the outer loop to a user-defined function, a mechanism to maintain the state between function calls is therefore required. This is achieved using the global `_record` variable into which the data generated during a cycle is stored. This data serves two purposes. Firstly it allows us to extract the previous state of the system during the subsequent step. Secondly, by returning this variable to the user-defined function, we expose useful data to the user for statistical purposes. The use of a dictionary rather than use a series of global variables, was simply to encapsulate all this information in one variable.

This had a repercussion for the algorithm in the main loop as resetting variables such as $[A]_{-1}$ and the action selection strategy at the end of a problem would have provided inaccurate information to the user. Therefore the code involved in readying the system for the next step was moved to the beginning of the cycle.

8.2 Testing

Python's standard library includes several utilities for unit testing. In order to test the low-level procedures in this suite we have used the `unittest` module. By subclassing the `TestCase` class, a set of unit tests can built up to assert statements about each procedure.

For example, to test the `couldSubsume` sub-procedure, that tests whether a classifier can legally subsume another, the following test case was built:

```
class CouldSubsumeTest(unittest.TestCase):

    def setUp(self):
...
        self.cl = Classifier([...])
        self.cl.exp = 21
        self.cl.e = 9.0

    def testCorrectCase(self):
        self.assert_(xcsp.couldSubsume(self.cl),
                     "Valid classifier not identified.")

    def testInexperiencedClassifier(self):
        self.cl.exp = 19
        self.failIf(xcsp.couldSubsume(self.cl),
```

```
                    "Inexperienced classifier not [...]")

    def testInaccurateClassifer(self):
        self.cl.e = 11
        self.failIf(xcsp.couldSubsume(self.cl),
                    "Inaccurate classifier not [...]")
...
```

A test case such as this was derived for each of the lower-level procedures in the algorithm. By "black box" testing each of the sub-procedures we can hope to achieve an accurate program. The inputs for each test case was derived by considering all possible pre-conditions (the states of the input variables for example) and checking that the correct post-conditions were created. However due to the non-deterministic nature of much of the algorithm it is not always so easy to create a reproducible test for each of the higher-level functions. For example, to test the `applyMutation` sub-procedure, it was necessary to change the value of the parameter μ first to 0 and check that no mutation occurred, then to 1 and check that mutation always occurred.

Once we had achieved a working set of functions, the system could be tested as a whole in order to test the higher-level functions (such as RUN GA and RUN EXPERIMENT). The results of this are described in Chapter 12.

Chapter 9

Object-oriented XCS

We now move on to develop the working ProceduralXCS implementation into an object-oriented one.

9.1 Candidate Classes

When designing an object-oriented system it is often useful to highlight the nouns in the particular domain one is working on and using them as a basis for the objects in the design (Abbott 1983). This was the case when designing the XCS component; the following candidate classes were found. The merits of implementing each of these candidate classes could then be weighed against the potential for causing inefficiency and over-complication.

- **Classifier System**: A top-level container class.
- **Classifier Set**: A classifier container (e.g. $[P]$, $[M]$ and $[A]$)
- **Classifier**: An individual in the population.
- The **Prediction Array** (PA)
- The **Genetic Algorithm** (GA)
- **Roulette Wheel**: A selection device used in selecting classifiers to replicate and delete.
- **Learning Parameter**: A system-wide value used in the algorithm.
- **Situation**: The percept sent from the environment (σ).
- **Reward**: The reward sent from the reinforcement program (ρ).
- **Condition**: The pattern matching attribute of a classifier.
- **Attribute**: An individual allele in the condition of a classifier.
- **Action**: The proposed action of a classifier.

9.2 `Classifier` Class

Probably the most straight-forward decision was to implement a `Classifier` class – it is so fundamental to the algorithm that we have already implemented a data storage version of this class in the ProceduralXCS module.

There were a variety of options for the composition of this class; mainly whether to use the built-in data types for all of its attributes or to implement condition, attribute and action classes (or a mixture of both). For a number of reasons, the former option was chosen. Firstly, it was decided not to use an action class as there did not seem to be any suggestion that XCS research will start to diverge from the convention of using an integer to refer to an action. Therefore we could see no real gains for the cost of the added complexity of a separate class.

There was greater feasibility in implementing a condition class and possibly an attribute class. Such a classes could encapsulate all the operations related to the condition's encoding and separate it from the other classifier functionality. This cleaner division is of benefit to a Developer wishing to implement a new encoding for their system. However, once again the added complexity of these classes was thought to outweigh the benefits that it would bring. Primarily, allowing the user to subclass their own condition and attribute classes would require a complex system of "builder" methods to construct the classifiers correctly. This may have been worthwhile if the class contained many other types of operations, unrelated to the condition. However there are actually very few; related to fitness calculation and subsumption. Therefore the `Classifer` class contains only built-in data types and encapsulates all the "classifier-level" operations. The condition is represented in the same way as in the ProceduralXCS module; as a list.

9.3 `XClassifierSystem` & `ClassifierSet` Classes

The need for a container for the whole system and a container for sets of classifiers was similarly obvious. However the way in which they were implemented—as built-in data types or classes—and how the functionality was encapsulated between the system and the set concepts was an important design decision.

One approach would be to store Classifier objects in a list container. This way, all the procedures from the algorithm relating to sets of classifiers would be in a classifier system class. This class performs the necessary append, remove, count and iterative operations on the lists. The other approach would be to make a classifier set class that encapsulates the set-level procedures itself.

After some deliberation it was decided to go for the latter approach and create both a `XClassifierSystem` class and a `ClassifierSet` class. The main reason for this choice was the ease in which the procedures appear to fall into these three levels of operation: system, set and classifier.

9.4 Algorithm Encapsulation

At first glance, a logical separation of the algorithm's procedures is as follows:

System	Set	Classifier
run[Experiment]	generateMatchSet	doesMatch
selectAction	generateCoveringClassifier	applyCrossover
runGA	generatePredictionArray	applyMutation
	generateActionSet	deletionVote
	updateSet	doesSubsume
	updateFitness	couldSubsume
	selectOffspring	isMoreGeneral
	insertIntoPopulation	
	deleteFromPopulation	
	actionSetSubsumeInsert	
	doActionSetSubsumption	

There are however some subtleties to be considered before we can accept this design.

Environment and Reinforcement Program References

Firstly, in order to separate the environment from the classifier system it was decided that the system should not keep references to the environment or reinforcement program. Instead, the information is to be fed to the system as method arguments. There are three pieces of information required by the classifier system on each cycle of the algorithm: σ, ρ and EOP. Unfortunately only σ can be supplied at the start of the `run` method and ρ and EOP are a consequence of action selection. The solution chosen was to separate the method into `run` and `update` phases.[1] σ is supplied to the `run` method which represents the performance component of the algorithm, and an action is returned. ρ and EOP are supplied to the `update` method which represents the reinforcement and discovery components. This neat separation of the phases of the algorithm comes at the cost of delegating the responsibility of calling these methods at the correct time to the user-defined top-level function.

Prediction and Fitness Updates

Next, the `updateSet` procedure could be described as being a classifier-level operation as it loops through each classifier in turn and updates its attributes directly using formulae that would be better suited to the `Classifier` class. Also, the use of the classifier's attributes directly increases the coupling between the two classes which is not a desirable feature in a class structure. Consequently, three methods were added to the `Classifier` class to update prediction, fitness and action set size. And the classifier set procedure was left only with the responsibility of calling these methods.

Crossover and Mutation Operators

A third consideration is the class of operation that the `applyMutation` and `applyCrossover` procedures fall under. Where it might be logical to use the

[1] Another solution would have been to move the update phase to the start of the method and supply the method with the reward from the previous step and the next situation. However this proved unintuitive and confusing, especially in multi-step problems where the so-called previous Action Set was in fact two steps old!

following code to mutate a classifier:

```
c1.mutate(s)
```

The same convention to perform the crossover operation, would be:

```
c1.crossover(c2)
```

This second use does not work as well as the first because it implies that there is a difference between the effect on the two classifiers – should you use the crossover method of $c1$ or $c2$? It therefore seems that the crossover operator would fit better as a class method, taking as parameters the two classifiers to operate on. Because the mutation operator is so closely related, it would also follow that it should be made a class method:

```
Classifier.crossover(c1, c2)
Classifier.mutate(c1, s)
```

Creating Covering Classfifiers

The `generateCoveringClassifier` procedure is the one procedure defined at the set level—rather than classifier level because it involves creating a classifier—that directly manipulates the condition string and so is affected by any alterations in its encoding. Consequently it was decided, as with the updating of the set, that the procedure should be separated between the two classes. The majority of the work to be done in the constructor of the `Classifier` class by specifying the current value of σ. The set only needs to decide on an action, instantiated a the new classifier object then do the necessary insertion and deletion.

Also, in this version we also introduce a second covering operation that was introduced by Wilson in his 1995 paper but not included in the algorithmic description. This covering is provided to prevent the system becoming stuck in a loop and is invoked if the system's combined Match Set prediction falls below a certain proportion (ϕ, set to 0.5 by default) of the average prediction of the Population. This is effective because the discounted payoff will cause the prediction of the classifiers to fall. Introduction of a new classifier, with a different action, should help break this loop – if not the first time, eventually a randomly selected action will work.

The second form of covering shares the same functionality for both the covering itself and the resulting classifier's insertion into the Population. As the functionality for creating the classifier is now encapsulated in the constructor of the `Classifier` class, it therefore made sense to abstract the entire process into a `addCoveringClassifier` function in the `ClassifierSet` class. This function selects an action from unused actions in the set, instantiates a classifier, then inserts it into the set (with any necessary deletion).[2]

[2] For extensibility the function takes an optional argument as the list of actions to choose from – the unused actions of the set being its default value.

Pluggable System-level Functions

The `XClassifierSystem` class contains two important functions other than `run` and `update`. Action selection and the GA are both key pieces of the XCS algorithm that are likely to be targeted by Developers wishing to alter their functionality. Therefore they have been written without reference to the system object that encloses them; all the required data is passed into the functions as arguments. In this way the functions can be easily replaced with customised ones either dynamically (at run-time by replacing the function variable in the XCS object) or statically (at compile-time by subclassing `XClassifierSystem`).

Also, the functionality that chooses the strategy for selecting an action (i.e. explore or exploit) has been separated from the SELECT ACTION procedure into a the `useExplore` function. This explicitly exposes the strategy selection mechanism and makes it easily accessible for adaption in the same way as above.

Insertion and Deletion from Sets

As a final consideration, the mechanisms for inserting and removing classifiers from the system needed to be addressed. If we wish to add a new classifier to the Action Set $[A]$ (during covering for instance), we need also add the classifier to the superset of $[A]$ – the Population $[P]$. Otherwise the classifier will be lost when $[A]$ is regenerated. Ideally we would prefer not to have to pass references to $[P]$ explicitly as arguments to methods as this is rather cumbersome. We therefore need to relate the sets to one another with a reference to their parent (the superset from which they were created). This linked list structure allows us to perform a cascading insert or deletion when required. However, there are many different scenarios for insertion and deletion in the algorithm; some require this cascading effect, others do not; some require subsumption and equality checks, others do not. Consider the following insertion scenarios. (A similar set of scenarios also exists for deletion.)

1. `runGA`: A new classifier is created and can be added to the set and its superset if it does not match, or get subsumed by, any members of the set.
2. `doActionSetSubsumption`: A new classifier is created and can be added to the set and its superset if it does not match any members of the set.
3. `addCoveringClassifier`: A new classifier is created and can be added to the set and its superset with no checks.
4. `generateMatchSet`: A subset is being created, classifiers can be added with no checks to the set alone.

To allow for the needs of these four different scenarios a series of insertion and deletion has been used:

- `insert`: Inserts a new classifier to a set and supersets if it doesn't match. The subsumption check is also done if specified as a parameter.
- `doDeletion`: Selects and deletes a classifier from the set and superset.
- `add` and `delete`: Adds and removes classifiers from the set and superset.
- `append` and `remove`: Basic insertion and deletion of classifier.

Revised Algorithm Encapsulation

So the revised method list for these three classes, with some rewording where appropriate, looks like this:

System	Set	Classifier
run	generateMatchSet	doesMatch
update	addCoveringClassifier	updatePrediction
useExplore	generatePredictionArray	updateActionSetEstimate
selectAction	generateActionSet	updateFitness
runGA	update	calculateDeletionVote
	doActionSetSubsumption	doesSubsume
	selectOffspring	couldSubsume
	insert	isMoreGeneral
	doDeletion	crossover
	add	mutate
	delete	
	append	
	remove	

Other Functions

In the procedural version there was functionality within some of the procedures that can now be extracted into separate methods. For example calculating total numerosity, average fitness and the average time since the last GA invocation. This allows us to remove the less important functionality and leave the basic algorithm in a purer form. Also, in the case of the total numerosity, this has the effect of removing duplicate code[3] and exposing the value for use in other places such as reporting.

The classes also take advantage of Python's "special" methods to enhance their behaviour. The `__str__` method is one example of a special method that determines how the object will be displayed as a string. It is similar to Java's `toSting` method, but Python calls the method automatically when the object is used in a string context. When used in a `print` statement, for example:

```
>>> cl = Classifier([...])
>>> print '%s' % cl
010010:0 => 0.0000 (k=0, f=0.0000, num=1)
```

The `__str__` should return a human-readable representation of the object. Another special method, `__repr__`, should return a more machine-readable representation that should contain all the data contained in the object. Therefore in the case of the `Classifier` class, this representation is a comma-separated string of values:

```
>>> cl = Classifier([...])
>>> print '%r' % cl
010010,0,0.0000,0.0000,0.0000,0.0000,0,11,1,1
```

[3] As it occurs both in UPDATE SET and DO DELETION.

In addition to `__str__` and `__repr__`, the `__eq__` and `__neq__` functions are implemented for the `Classifier` class to redefine the equality and inequality operators between `Classifier` objects:

```
if c1 == c2:
    # Classifier already exists, don't add it!
    ...
```

This achieves a more useful meaning of equally; that `c1` has the same condition and action as `c2`, rather than verifying if the two variables refer to the same object.

In the case of the `ClassifierSet` class, special methods allow us to make our set class act more like a container for the classifiers rather than having the classifiers simply being a property. For example, instead of accessing the first classifier in the Population using the expression `population.classifiers[0]`, we implement the `__getitem__` function to allow us to treat the set as the container and access the classifier thus: `population[0]`.

In addition, the `__contains__`, `__iter__` and `__len__` methods were also implemented for the `ClassifierSet` class to allow us to check whether a classifier is in a set; iterate through the set; and get the length of the set, respectively. In the example below, notice how an instance of our set class is now behaving as if it were a list:

```
if len(P) >= max:
    return False
else:
    for cl in P:
        if cl == child:
            return False
    P.append(child)
```

The set also contains a sorting method to rearrange the order the classifiers are displayed in. Lists in Python contain a `sort` method that uses the greater-than and less-than operators to sort its members into order. `sort` also takes as an optional argument, a custom comparison function to use instead of the inequality operators. This function should take two members of the set as arguments and return -1, 0 or 1 depending on whether the first member is less than, equal to, or more than the second. The `sort` function was implemented for the `ClassifierSet` in the same way, but if no sorting function is supplied, the `default_order` class function of the `Classifier` class is used. This function sorts by accuracy, then numerosity and then experience.

Added to the `XClassifierSystem` class was a reset function that reverts the system back to the state it was in before any steps were run. To enable an initial population to be reused, a deep copy of the set was taken in the constructor. Upon being reset the system takes another copy for the next experiment.

The `XClassifierSystem` will also maintain counters on the number of problems and steps it has taken. These can then be used by the controlling class for statistical purposes or to decide upon termination. The following counters will be stored:

- `explore_step_count`

- `exploit_step_count`
- `explore_problem_count`
- `exploit_problem_count`
- `current_step_count` (the number of steps taken on the current problem)

Finally, in order for the class of classifier set to be changed easily via subclassing, the `getDefaultClassifierSet` function was added. This simply moves the functionality involved in instantiating the set away from the constructor so that it can be easily overridden.

9.5 Learning Parameters

The learning parameters in this implementation were handled as class variables belonging to either the `XClassifierSystem`, `ClassifierSet` or `Classifier` class. Not only does this produce neater code when accessing the values, compared to accessing them from a dictionary, it also has the advantage of giving the parameters global scope. The parameters have been separated between the classes according to the procedure that it is primarily used for. This was not strictly necessary as, due to this global scope, they could have all belonged to one class. However, it seemed to help distinguish between the parameters and their uses to separate them as shown in Figure 9.1.

In other languages class variables cannot be defined at run-time, so the values of parameters would have to be static (defined at compile-time). However, in Python everything is an object and class variables can be modified using the `class` objects. This means that the system will still be able to change the default values for the parameters at run-time. If we wish to read them from a text file, for example. However, there are some parameters (shown in Figure 9.1 in italics) whose values are dependent on the environment and so could not be given default values. The value of these parameters is `None` until they are set by the `XClassifierSystem` constructor using values passed to it from environment.

This version introduces four new parameters: `do_performance_covering` and `phi` control the new covering mechanism described earlier in the section; also `X_reduct_e` and `X_reduct_f` were added to parameterise the reduction in the error and fitness values given to offspring.

9.6 Selection Mechanisms

The algorithm contains several selection processes:

- **Strategy**: Explore strategy is chosen with a probability P_{exp}, otherwise exploit is chosen.
- **Action (Explore)**: Action is randomly selected from the PA.
- **Action (Exploit)**: Action with maximum system prediction is selected from the PA.
- **Replication**: Classifier is selected for replication with probability proportional to fitness.

System		
`g`	γ	Discount factor for payoff.
`T_GA`	θ_{GA}	Experience threshold for GA.
`P_exp`	P_{exp}	Probability of choosing explore strategy.
Set		
N	`N`	Maximum macroclassifier population.
`phi`		Used in performance cover calculation.
`T_mna`	θ_{mna}	*Minimum number of actions before covering occurs.*
Classifier		
`n`	n	*Number of actions.*
`L`	L	*Length of a situation.*
`B`	β	Learning Rate.
`p_1`	p_1	Initial prediction of a new classifier.
`e_1`	ϵ_1	Initial error of a new classifier.
`f_1`	f_1	Initial fitness of a new classifier.
`P_X`	χ	Probability of crossover occuring.
`X_reduct_e`		Proportion of error to pass to offspring.
`X_reduct_f`		Proportion of fitness to pass to offspring.
`P_mu`	μ	Probability of mutation occuring.
`a`	α	Used in accuracy calculation.
`nu`	ν	Used in accuracy calculation.
`e_0`	ϵ_0	*Threshold below which error is considered zero.*
`e_0_d`	ϵ_{0_δ}	Proportion of maximum reward to set ϵ_0.
`P_wc`	$P_\#$	Probability of a wild card being used in covering.
`d`	δ	Used in deletion vote calculation.
`T_del`	θ_{del}	Experience threshold for deletion.
`T_sub`	θ_{sub}	Experience threshold for subsumption.

In addition, the following boolean values are defined for `XClassifierSystem`:

`do_GA_subsumption`	Whether to check parents for subsumption of offspring after GA.
`do_AS_subsumption`	Whether to perform a periodic subsumption check in $[A]$; subsuming at most one classifier per timestep.
`do_GA_subsume_insert`	Whether to check $[A]$ for subsumption of offspring after GA, before insertion into the population.
`do_performance_covering`	Whether to perform covering if the total prediction for $[M]$ falls below ϕ of the average prediction for $[P]$ (Wilson 1995).

Figure 9.1: The Learning Parameters used in the framework.

- **Deletion**: Classifier is selected for deletion with probability proportional to deletion vote.

We could implement a class with a standard interface and allow these selection mechanisms to become objects. These objects could then be pluggable in the `XClassifierSystem` to allow the user to easily change the way these selections are done. In order to do this we add an abstract base class, `SelectionMechanism`, with three methods: `add`, `select` and `reset`. From this we simply need to derive concrete classes to produce actual selection mechanisms; `RouletteWheel` or `Tournament` for example.

We have so far been vigilant in maintaining the simplicity of the code and it could well be argued that the introduction of this class is an unnecessary abstraction. This is particularly the case for the first three selection processes described above; to use a selection object is likely to be an unnecessary convolution as it would involve recreating the Prediction Array in the object. However, for replication and deletion, the process demands that we construct some sort of data structure to add the candidates to – it may just as well be in our custom object.

The `SelectionMechanism` class was thus implemented and subclassed to form the `RouletteWheel` class for use, by default, in replication and deletion. The user can modify this selection object dynamically using properties of the `ClassifierSet` object.[4] Alternatively, they can subclass the set class and override the `getDefaultReplicationSelector` and `getDefaultDeletionSelector` methods.

9.7 Other Classes

At the beginning of this section we identified many of the candidates for classes in the framework. Of these, the Prediction Array and Genetic Algorithm concepts remain to be examined. Both have potential to be implemented as classes in this framework, however both were eventually rejected.

Despite having relatively few classes, Butz includes a Prediction Array class in his Java implementation. Conversely Barry encapsulates the Prediction Array with the Match Set functionality in his `SystemPrediction` class. A class could be created that encapsulates the array itself and the functionality for its construction and for selection from it. However this seemed an unnecessary dispersion of the original algorithm and would require the class to be closely coupled to other elements of the framework; with knowledge of the action selection strategy, Action Set and classifier attributes. In the case of the selection mechanism, the class was justifiable because it only removed code that was unrelated to the XCS algorithm itself (i.e. the mechanics of selecting an individual object from a set of weighted objects). Its functionality was also much less related, and so less tightly coupled, to the actual classifiers.

During the initial design stages the GA was a strong contender due to the fact that they are part of the wider research area of "Evolutionary Computing". We can therefore envisage new developments in genetic algorithms from this field being applied to the GA in our XCS framework. The GA is concerned with

[4] The same object is passed to any subsets created from the set, so the properties (`replication_selector` and `deletion_selector`) only needs to be set in the population set.

three activities of the discovery phase: selection of parents, and the crossover and mutation of offspring. We could therefore have constructed the GA as a composite class (Gamma, Helm, Johnson & Vlissides 1994) consisting of each of these operations either as three further classes or three function objects. A single GA object, whatever its internal functionality, could then be plugged into the system. It cannot be argued that this is not very neat solution, however to allow the GA to manipulate the classifiers' conditions (for crossover and mutation) would violate the design rules we have so far adhered to. Therefore the use of a GA class was abandoned and its functionality dispersed; with top-level operation and offspring selection in `XClassifierSystem`; and crossover and mutation in the `Classifier` class.

Chapter 10

Experimenting in the Framework

Having implemented both the environment and classifier system, the framework is near completion and we can begin setting up experiments. This section looks at exactly what is involved in creating experiments and how the framework can provide an `Experiment` class to help.

10.1 What is an Experiment?

Assuming we use the object-oriented version of the XCS algorithm, a basic experiment can be easily created:

```
1 env = BooleanMultiplexer(2)
2 xcs = XClassifierSystem(env.n, env.L, env.max_reward)
3 while xcs.exploit_problem_count < 5000:
4     s = env.getSituation()
5     act = xcs.run(s)
6     r = env.getReward(act)
7     xcs.update(r, env.EOP)
8     env.executeAction(act)
9     env.reset()
```

The first two lines instantiate the environment and classifier system objects which are then used in the main loop to run the experiment. First the current situation is retrieved (line 4) and fed into the classifier system to select an action (line 5). The reward for the proposed action is then stored into `r` (line 6) which is used to update the classifier system (line 7). Finally the action is executed (line 8) and the environment is reset for the next problem (line 9).

We could have constructed an initial population to be used by the classifier system by passing it to the constuctor. We could also have made changes to the values of the learning parameters before the experiment begins, either by making the changes statically (as below) or by reading some form of representation from a text file.

```
Classifier.P_mu = 0.05
```

The next step in the development of this experiment code would be to add some feedback to the user. The simplest form of feedback would be to report the resulting population of classifiers to the user:

```
for cl in xcs.P: print cl
```

We may also wish generate statistics from each step of the algorithm – such as a moving average of the population of macroclassifiers.

```
[...]
population = []
while xcs.exploit_problem_count < 5000:
    [...]
    if xcs.explore == False:
        population.insert(0, len(xcs.P))
        if len(population) > 50: population.pop()
        population_sum = 0
        for p in population: population_sum += p
        print population_sum / len(population)
```

Of course in both cases the output could easily be written to text files rather than the screen. This would allow it to be analysed by external applications and processes.

We may also wish to re-run our experiment a number of times in order to verify the data that is collected or perhaps iteratively modify the value of a learning parameter.

10.2 The `Experiment` Class

The `Experiment` class is designed as the default controlling class for the framework and allows the user to customise their experiments in all of the ways described in the previous section.

However, before we move on to a more detailed description of this class, it needs to be emphasised that this is just *one* method of implementing an experiment. The class hides all of the code previously described and gives the user a simple interface to provide the inputs of an experiment—an environment at least—and receive its outputs. It is expected that the class will not be flexible enough to satisfy the needs of all users; they will need to write their own programs similar to the code described in the previous section.

Class Deconstruction

The class is created by supplying an environment object and four other optional elements: a reinforcment program object (if seperate from the environment); a

classifier system object (if the standard `XClassifierSystem` is not to be used); the location of an XML file containing learning parameters; and a logging object.

The XML parsing uses the `Sax` module (one of Python's standard library modules). The file itself should be structured in the following way:

```
<parameters>
    <class name='Classifier'>
        <param name='B'>0.2</param>
    </class>
    <class name='ClassifierSet'>
        <param name='N'>800</param>
    </class>
    <class name='XClassifierSystem'>
        <param name='g'>0.71</param>
    </class>
    <class name='my_classifier' module='my_mods'>
        <param name='j'>8</param>
    </class>
</parameters>
```

As this snippet shows, not all parameters have to be specified; default values are used if not value is given. Also, new classes can be specified in the XML file and their class variables will be updated accordingly. If a custom class, such the `my_classifier` class, is found, it is imported from the specified module. Therefore if the class if not found, a warning is generated and no updates are done. Similarly if the specified parameter is not found, a warning is produced and the default value is used.

These warnings, and others that can be generated by the `Experiment` class are reported using Python's `logging` module. The built-in `Logger` class is used to configure the destination of different levels of system messages. If no logging object is specified to the constructor, it configures an object that displays errors and warnings to the screen. The user may wish to configure their own object to display information and debug messages as well, or to place the messages in a text file or streamed to another destination. An initial population can also be specified using the `setInitialPopulation` function. The set is copied so that the experiment can be reset and run again if required.

To run an experiment, the user simply calls the `run` method. This method takes as an optional argument the number of times to repeat the experiment. Termination of an experiment is configured beforehand using one of the following functions:

- `setTerminationStepCount`: Sets a maximum number of steps after which the experiment will terminate. The user can also specify whether these steps are explore, exploit or both.
- `setTerminationProblemCount`: Sets a maximum number of problems after which the experiment will terminate. Again, the user can specify whether these problems are explore, exploit or both.
- `setTerminationFunction`: Registers a function that is called at the start of every step with a reference to the classifier system as a parameter. If the function returns `True`, the experiment terminates.

The default action is for termination after 5000 problems in total. If more than one of these are set, the most recent critereon is used. In fact, the latter function sets the experiment's private `_terminate` variable to the specified function; the first two functions set the same variable with predefined functions for counting steps and problems.

In order to allow other processes to access information from the experiment while it is running, the user can register their own functions to be triggered on certain events. `registerStepListener`, `registerProblemListener` and `registerExperimentListener` can are therefore used to add as many functions to each event as the user requires. The listening functions for the first two events must take two arguments: the number of the experiment that is currently running; and a reference to the classifier system. Experiment listeners must take the experiment number, the time (in seconds) the experiment took to run and the final set of classifiers.

As reporting is such a common task for an XCS experiment, the `Experiment` class contains the functionality to generate reports for the statistics and results of an experiment (or series of experiments). For efficiency reasons, these reports are only generated if the `setResultsReport` or `setStatisticsReport` functions are called to specifiy a file location for the report. The results report simply writes the machine-readable representation (as a comma-seperated list) of each macroclassifier at the end of an experiment to the file. The statistics report is a more complex set of figures that are generated at the end of each problem. These are moving averages over a period of 50 problems (by default) and measure performance, error and population size. However performance is typically measured differently for single- and multi-step problems, therefore four values are calculated:

- **Step Count**: The average number of steps took to solve the problem. On single step problems this is obviously always 1, but on multi-step problems this is used as the performance indicator.
- **System Performance**: The average reward earned on each step. On multi-step problems this means little, however on single-step problems this is used as the performance indicator.
- **System Error**: The average difference between the reward earned and reward predicted.
- **Macroclassifier Population**: The average number of macroclassifiers in the Population.

It is unfortunate that an extra performance value will always be calculated, but the framework does not distinguish between the two problem types. The user is therefore free to select the meaningful values from the four supplied.

The reporting functionality has two solutions for dealing with successive experiments. The user has the option to either append data to the same file or generate multiple files for each run. This is done using the `setReportMode` function to specify whether this "append mode" should be used or not. The function also takes an optional second string argument which has different uses depending on the mode being used. If append mode is used, the string defines the separator line to be printed between runs. If append mode is not used, the string defines the suffix (i.e. the text between the name of the report and the

extention) to be used for each file. The separator string can contain the "%n" placeholder which is replaced by the experiment number during the experiment. This must be included when append mode is off in order to generate unique file names.

Finally, the `Experiment` class also records the time, in seconds, that is taken to run each experiment. This is crudely obtained by calculating the difference in time between the start of the experiment and the finish (excluding reporting). This means that it does not take into account external influences such as background processes starting or stopping during the experiment and effecting performance. However it sometimes serves as a useful indicator, especially if averaged over a number of runs. The time is reported in the default results file and is passed into experiment listeners. A list of times can also be retrievd using the `getExperimentTimes` function.

10.3 Creating an Experiment

To create a simplistic experiment using this developed class, we now only need a few lines of code:

```
env = BooleanMultiplexer(2)
exp = Experiment(env, parameters='params.xml')
exp.setTerminationProblemCount(5000, explore=False)
exp.setResultsReport('results.csv')
exp.setStatisticsReport('stats.log', explore=False)
exp.run(10)
```

Two experiment programs have been written and included in Appendix G. These create an `Experiment` object interactively using input from the command prompt. Output from the Multiplexer program (mux.py) is shown in Figure 10.1.

```
  Boolean Multiplexer Environment
  ===============================

>> Enter number of experiments to run [1]: 10
>> Enter the size of multiplexer, k [2]:
>> Enter number of problems to test [5000]:
>> Count Exploit problems? [y]:
>> Count Explore problems? [n]:
>> Enter file for system parameters [None]: params.xml
>> Enter output file for results [None]: results.csv
>> Enter output file for statistics [None]: stats.log
>> Record Exploit problems? [y]:
>> Record Explore problems? [n]:

  Running Experiments ... (Press Ctrl-C to cancel)
  Experiment 1 Complete. (13.489000 secs)
  [...]
  Experiment 10 Complete. (16.654000 secs)

  Complete.
  (Total: 143.366000 secs, Average: 14.336600 secs)

>> Press any key to exit.
```

Figure 10.1: Output from the interactive Multiplexer program (mux.py)

Chapter 11

Framework Distribution

Having created the classes that make up the framework they need to be structured into a package that can be easily distributed.

11.1 Code Structure

When writing large programs in Python, such as this framework, the code is separated into modules and packages. Unlike languages such as Java, Python files does not usually just contain a single class definition (unless a single class is the most logical content). Instead, the contents of one file is known as a module and usually contains a range of associated classes, methods and global variables. The classes making up the framework that we have described in the previous sections have been organised into the following modules:

- **Experiment**: `Experiment` (abstract) class.
- **Environment**: `Environment` and `ReinforcementProgram` classes.
- **BooleanMultiplexer**: `BooleanMultiplexer` class.
- **Woods**: `Woods` class.
- **ProceduralXCS**: `setup` method, XCS algorithm methods and `_t`, `_env`, `_rp`, `_params` and `_record` global variables.
- **test_ProceduralXCS**: Test case classes for ProceduralXCS module
- **XClassifierSystem**: `XClassifierSystem`, `ClassifierSet` and `Classifier` classes.
- **test_XClassifierSystem**: Test case classes for XClassifierSystem module.
- **SelectionMechanisms**: `SelectionMechanism` (abstract) class and the `RouletteWheel` and `Tournament` classes.

In a similar way to other object-oriented programming languages, Python uses packages to create namespaces to avoid variable naming clashes and provide a means to logically group modules. Packages are simply directories containing

the module files (or further subpackages) and a (usually empty) Python file named "__init__.py" to signify to the Python interpreter that the directory is a package.

The framework has been organised into the packages shown below. This was done simply to separate the XCS functionality from the implemented environments and test suites. It is expected that in future versions, the environment package will be augmented with other useful environments and any modifications to XCS functionality (e.g. XCSm, XCSC, etc) will either be placed into the root or into further subpackages.

- xcs/
  - Experiment
  - Environment
  - ProceduralXCS
  - XClassifierSystem
  - SelectionMechanisms
  - environments/
    - BooleanMultiplexer
    - Woods
  - test/
    - test_ProceduralXCS
    - test_XClassifierSystem

Python uses the same dotted notation as Java when referring to packages and modules. We therefore require the following import statements at the top of a typical experiment:

```
from xcs.Experiment import Experiment
from xcs.environments.Woods import Woods
```

11.2 Other Files

The framework will als be distributed with an installation script, documentation, interactive scripts for the Multiplexer and Woods experiments and a "Read Me" file containing installation, copyright and contact information.

The installation script was generated using Python's `distutils` module. When run with the "install" option it installs the relevant source files into the library of the user's Python installation. There is also a self-contained installation executable file that was also generated by the `distutils` module.

The documentation has been generated using the Epydoc module; an automated documentation tool for Python. This uses a structured commenting system for all modules, classes, functions, properties and class variables, and produces documentation in a variety of formats. Contained in the package are both PostScript and HTML versions of this documentation.

11.3 Package Name

Finally, the package was named "pyXCS". This follows the Python package naming convention of using the "py" prefix. This name is not mentioned in any of the code, but is used simply to distinguish the package from other XCS implementations and other Python packages.

Part IV

Results

Chapter 12

System Testing

This section documents the results of the system testing that was carried out on both the object-oriented and procedural versions of the algorithm. We look at both the accuracy of the results that it generates (i.e. the final population of classifiers generated in an experiment) and the system's performance in terms of the problem-by-problem statistics generated during an experiment. We also use Butz's Java XCS implementation to generate data that we can compare with our own results.

As it is based on the same algorithm, Butz's Java XCS should provide a useful comparator. However, in order to compare the statistics we had to setup an experiment with modified reporting function to match the Java version. The main difference is that the default report for the `Experiment` class reports a moving average at the end of each problem, whereas Butz produces an average value every 50 problems. The values used in the report also had to be scaled in the same way as Butz's results were. In addition, the mechanism for selecting whether to use exploration or exploitation was also modified to match the Java implementation so that the strategies alternated on each problem, rather than being selected randomly.

12.1 Boolean Multiplexer

Figure 12.1 shows the final population in the 6-Multiplexer environment of a typical experiment run in the object-oriented framework. The classifiers shown in bold are the 16 maximally general rules that are required to accurately predict an action.[1]

The rules are not simply present in the population, they make up the 16 most numerous classifiers. The results in Figure 12.1 show these 16 accurate macroclassifiers in the process of taking over the population. The existence of other rules is due to the GA's continual generation of new classifiers. However the experience and numerosity of these rules is small, suggesting that they will soon be subjected to deletion.

An experiment was set up using the 6-Multiplexer over 5000 exploit steps on the procedural and object-oriented versions, as well as Butz's Java-XCS. The

[1] For more information on these rules, see Appendix C (p. 84).

29 x 001###:1 → 1000	3 x 1####0:0 → 800
26 x 01#1##:1 → 1000	3 x #01###:0 → 269.054706
26 x 01#1##:0 → 0	3 x 1###1#:0 → 219.791209
25 x 11###0:0 → 1000	2 x 1###0#:1 → 252.4288
25 x 001###:0 → 0	1 x #00###:0 → 666.666667
23 x 10##1#:1 → 1000	1 x 1##10#:1 → 400
23 x 000###:0 → 1000	1 x 00####:0 → 412.896789
23 x 01#0##:0 → 1000	1 x #01#0#:0 → 416
22 x 11###0:1 → 0	1 x #1####:0 → 360
20 x 10##0#:1 → 0	1 x 1####0:1 → 81.92
20 x 01#0##:1 → 0	1 x 1####1:1 → 760.7936
19 x 11###1:0 → 0	1 x #1#10#:1 → 896.605164
18 x 11###1:1 → 1000	1 x #00###:1 → 528.930836
18 x 000###:1 → 0	1 x #0##1#:0 → 375.298327
17 x 10##1#:0 → 0	1 x 1##0#:0 → 733.340988
17 x 10##0#:0 → 1000	1 x #0#1##:0 → 671.296067
4 x #01#1#:0 → 0	5 x #0#10#:0 → 1000
3 x 1###00:1 → 0	2 x 1###00:0 → 1000
2 x #00#0#:1 → 0	1 x 11####:1 → 1000
2 x 1###11:1 → 1000	1 x #1#0##:1 → 0
1 x 100#10:1 → 1000	1 x 11#10#:1 → 1000
1 x 0#00##:1 → 0	1 x 1####1:0 → 0
1 x 1###11:0 → 0	

Figure 12.1: The macroclassifiers produced on a typical run using the object-oriented version of the Python framework. Out of the 45 that were generated on this run, the 16 most numerous (shown in bold) are the maximally general rules that are needed to accurately classify the data in the Multiplexer environment.

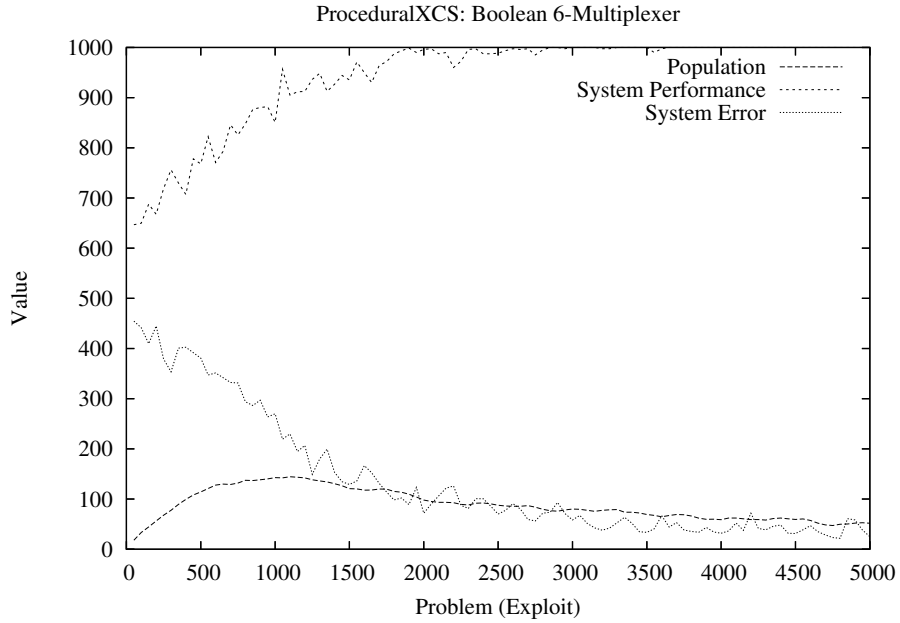

Figure 12.2: The averaged (and scaled) results from running the Boolean Multiplexer on the ProceduralXCS version of pyXCS. (Significant parameters: $N = 400$, $\mu = 0.04$.)

experiments used equivalent learning parameter values and were repeated 30 times. The averaged performance statistics are shown in Figures 12.2 – 12.4.

Looking at the statistics for the procedural and object-oriented algorithms; we appear to have achieved something close to the expected behaviour. The behaviour of the system's error and performance over time are as expected – error decreases while performance increases as the system develops better, more accurate rules. The population of macroclassifiers initially rises then gradually falls away. This is also the expected behaviour, as the number of different classifiers is initially high whilst the system is discovering and trying many different classifiers. However, gradually the accurate classifiers start to dominate the population, and the macroclassifier count falls as the numerosity of the accurate classifiers increases.

Although generally the Python algorithms exhibit the same behaviour as observed by the Java implementation, there are noticeable differences between the exact paths of each of the lines in the three experiments. T-tests taken at four samples (250, 1000, 2000 and 5000) over the time period confirm that there are some statistically significant differences between the samples. Most notably between the procedural version and the Java implementation, and in the latter stages between the object-oriented version and the Java implementation. Further results are shown in Appendix E.

Finally, as an interesting demonstration, Figure 12.5 shows a typical run using the object-oriented version (with default reporting) on the 11-Multiplexer problem over 15000 exploit problems. It too shows the classifier system learning correct behaviour over time, with macroclassifier population dropping to around

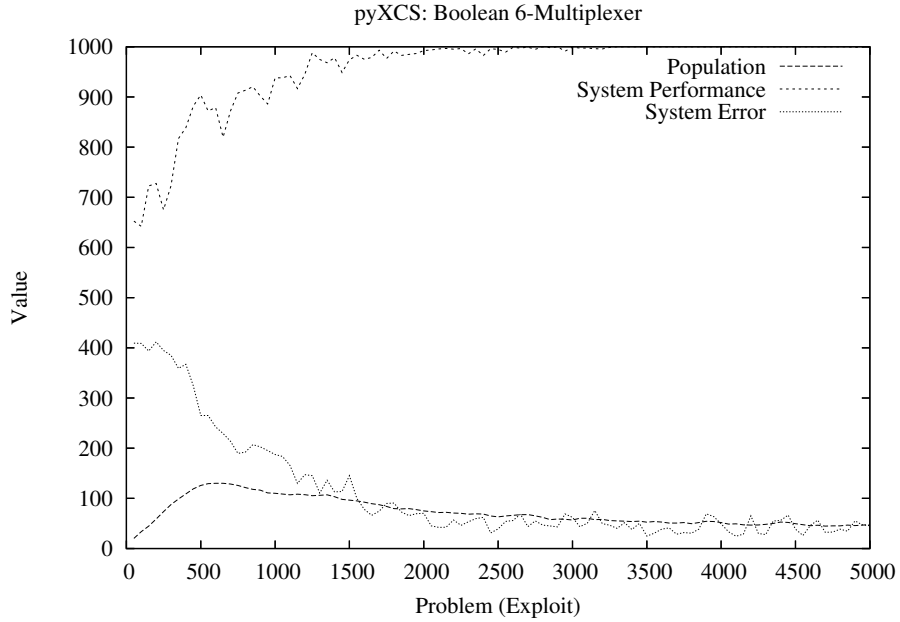

Figure 12.3: The averaged (and scaled) results from running the Boolean Multiplexer on the object-oriented version of pyXCS. (Significant parameters: $N = 400$, $\mu = 0.04$.)

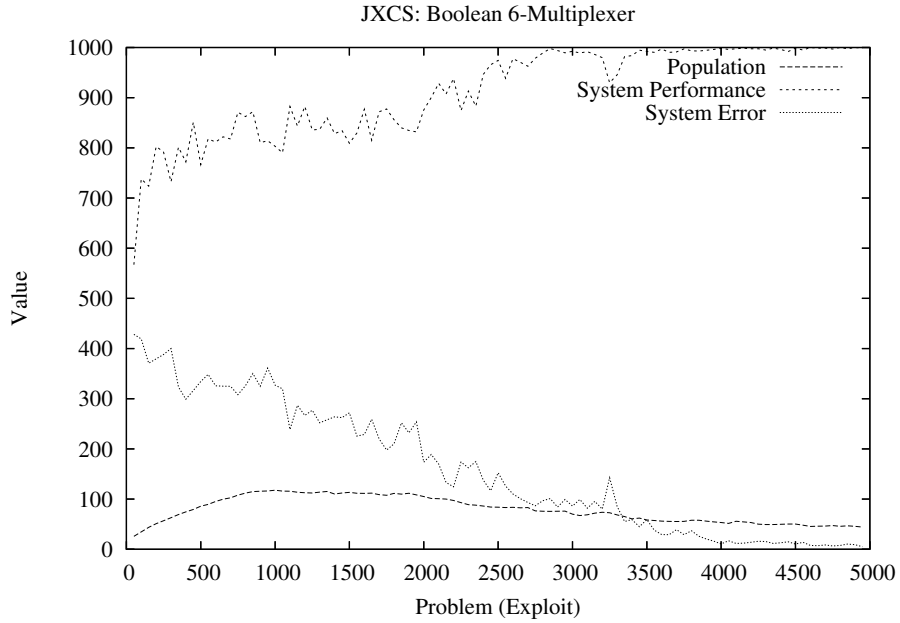

Figure 12.4: The averaged (and scaled) results from running the Boolean Multiplexer on Butz's Java XCS implementation. (Significant parameters: $N = 400$, $\mu = 0.04$.)

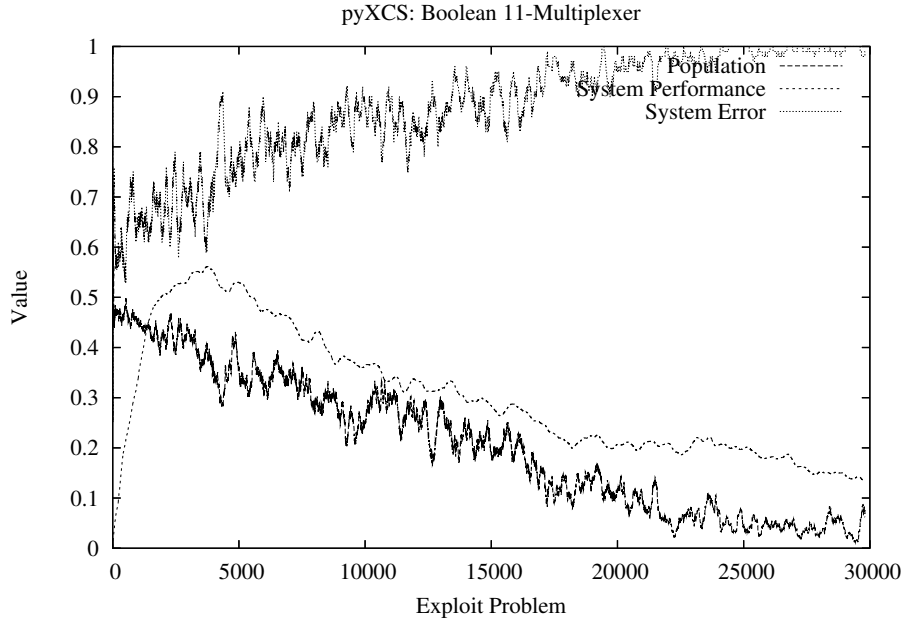

Figure 12.5: The averaged (and scaled) results from running the Boolean 11-Multiplexer on the object-oriented version. (Significant parameters: $N = 800$, $\mu = 0.04$.)

100 and performance and error tending towards the maximum and minimum, respectively.

12.2 Woods

The Woods1 maze was also tested in the three implementations under the same conditions as the Multiplexer[2] and the statistics are shown in Figures 12.6 - 12.8. However, these do not show the Python framework behaving in such as expected way.

The object-oriented version (Figure 12.7) does mirror the Java implementation's behaviour to some extent; we see the rise and fall in population and a leveling out of the number of steps taken. However the number of steps in the Python version fluctuates more than the Java version and averages around 2.5, whereas the optimum number of steps, as shown by the Java version, is around 1.7.

In the Woods environment, when we look at the final populations of classifiers produced we should see that if they are sorted in order of prediction, the classifiers are grouped according to how many steps away from the food the condition they match is. Therefore we expect to see a relatively small group that are one step away and so predict the maximum reward (in this case 1000); another group that are two steps away and predict the maximum reward dis-

[2] Although the system parameters were modified to set N as 800 and μ as 0.01 – a more standard configuration for the Woods environment.

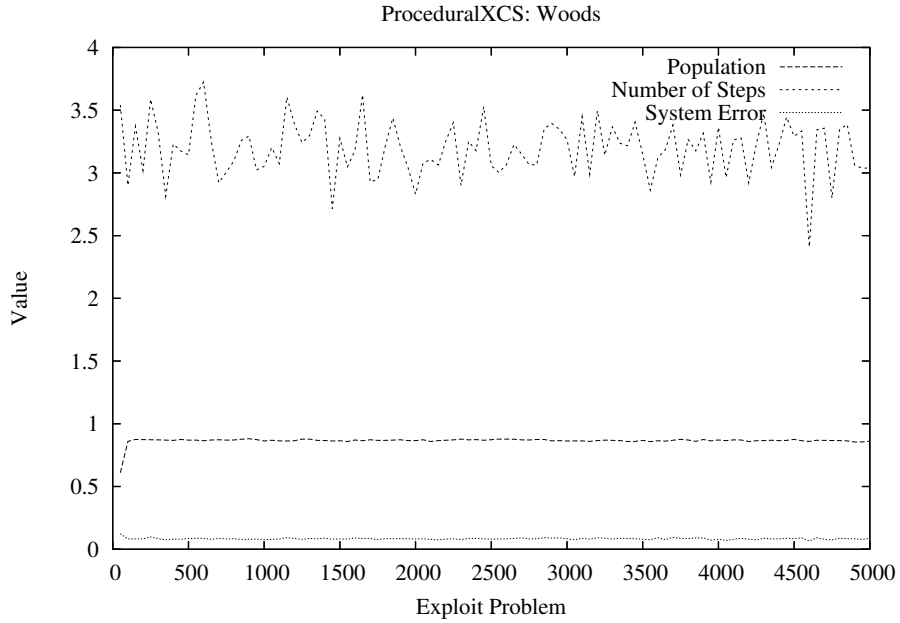

Figure 12.6: The averaged (and scaled) results from running the Woods1 environment on the ProceduralXCS version of pyXCS. (Significant parameters: $N = 800$, $\mu = 0.01$.)

counted once (around 709); and some that are three steps away and predict a twice-discounted reward (around 503). This is certainly the case with Butz's results; the vast majority of the classifiers fall into one of these three categories. It is also the case to some extent with the object-oriented version, although there appears to be more of a dispersal in the values of predictions which might account for the higher average step count of the Python version.

The procedural version, however, does not appear to be working as expected at all. Population does not fall away after reaching around 80% and the number of steps fluctuates dramatically with an average of just over 3 steps.

T-tests were conducted on the object-oriented version compared to the Java implementation in the same way as the Multiplexer. This found significant differences in half of the samples.[3] Despite many efforts to remedy the problems found in the multi-step environments, it was not possible to identify the exact cause of the problem. The code for running multi-step problems is conceptually the same as both Butz and Barry's implementation, yet fails to exhibit the correct behaviour. This is discussed later in Section 13.2 (p. 76.

[3] It was not thought neccessary to perform a t-test on the results from the procedural version as these results can be seen visually as being different.

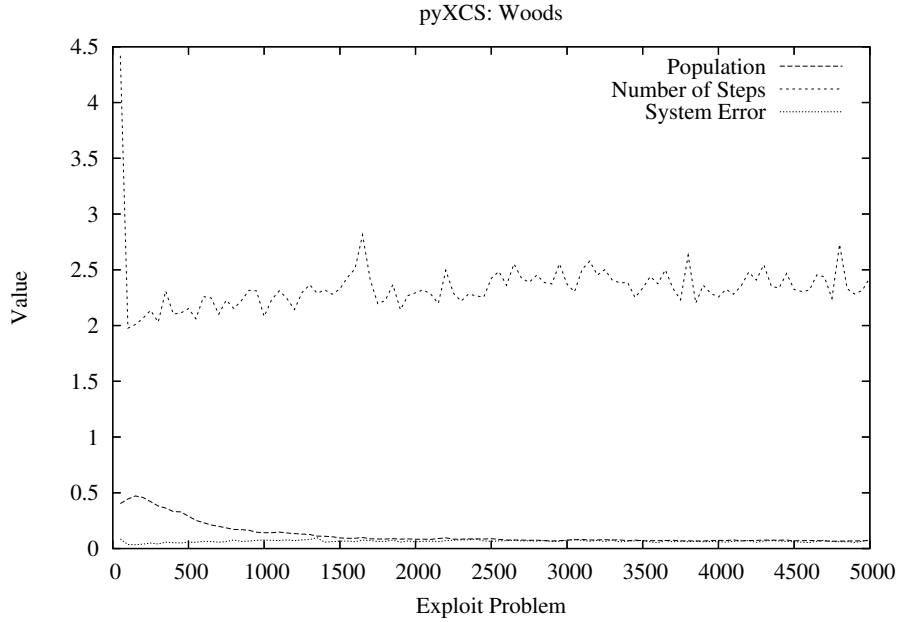

Figure 12.7: The averaged (and scaled) results from running the Woods1 environment on the object-oriented version of pyXCS. (Significant parameters: $N = 800$, $\mu = 0.01$.)

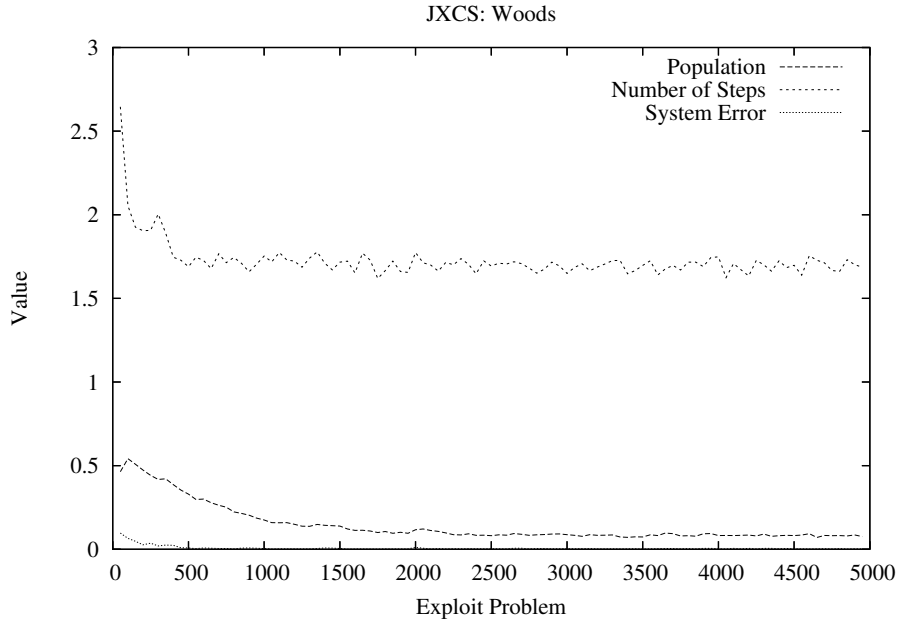

Figure 12.8: The averaged (and scaled) results from running the Woods1 environment on Butz's Java XCS implementation. (Significant parameters: $N = 800$, $\mu = 0.01$.)

Chapter 13

Conclusion

In the last section we looked at the performance of the system as an XCS implementation. We have found that the framework performs as expected for the single-step Boolean Multiplexer; it generates the correct end population and the performance is very similar to the Java version. However, the multi-step environments have shown to be more problematic to for Python algorithms. Whilst the object-oriented version does get some way to solving the problem, the procedural version does not exhibit the correct behaviour at all.

In this section we analyse and critique the system's overall effectiveness as a framework for development. In order to do this, we will be referring back to the formal requirements specification (see Appendix B, p. 81) that was developed during the design stage of the project. Results from this analysis can then lead to plans for future development of the framework.

13.1 Revisiting the Requirements Specification

Whilst developing the framework the more practical functional requirements[1] were used to guide development and were all met with little further comment needed:

- **Programming Language Constraints** (Requirements 1c – 1f): The framework was developed using just the standard libraries from Python 2.4 (the most recent at this time of writing) and written (with the justifiable exception of the ProceduralXCS module) in an object-oriented structure.
- **Documentation** (Requirement 2): An automated documentation tool was used to facilitate the documentation. This also meant that the comments used in the documentation accompanied each of the functions within the code itself as a prompt for those reading the source code.
- **Learning Parameters** (Requirement 3): Using class variables and a built-in XML parsing library these were easily modifiable.
- **Reporting** (Requirement 4): This was implemented according to the specification in the `Experiment` class.

[1] That is, those requirements that related to specific functionality and features of the framework.

- **Environment Creation** (Requirement 5): A lot of design effort was placed into the separation of the environment from the classifier system. Also, through the use of Python's multiple inheritance mechanism the conceptual separation of the environment from the reinforcement program was also achieved without the added complication of extra concrete classes.

The other requirements, however, do warrant further discussion and analysis.

Code Clarity (Requirement 1a)

During the development of the core classes (the system, set and classifier) the trade-off between clarity and efficiency was prominent. Some functions benefited from the transformation between a procedural and object-oriented structure. For example the `update` function of the `ClassifierSet` is reduced from 21 lines of code to 11. The code, that contained some relatively obscure calculations, is presented more clearly in its object-oriented form and gives a better overview of the update process.

```
1  accuracy_sum = 0
2  total_numerosity = self.total_numerosity
3  for cl in self:
4      cl.exp += 1
5      cl.updatePrediction(payoff)
6      cl.updateActionSetEstimate(total_numerosity)
7      accuracy_sum += cl.k
8  for cl in self:
9      cl.updateFitness(accuracy_sum)
10 if XClassifierSystem.do_AS_subsumption:
11     self.doActionSetSubsumption()
```

This function also shows an example of the clarity versus speed trade-off. Line 2 is used to prevent the total numerosity being calculated for each classifier. The resulting update would be the same if it were calculated each time, but it would take longer. It was decided for situations like this, Requirement 1b regarding speed and efficiency takes precedence over Requirement 1a regarding the emulation of the original algorithm.

Two features of the object-oriented approach stand out as having suffered from the transformation from their procedural equivalents. Firstly, by allowing the user to use any derived classifier and set class meant that the following lines of code were used to create their respective objects:

```
cl = self.classifier_type(t, act, s=s)
set = self.__class__(self.classifier_type, self)
```

Instead of the more readable:

```
cl = Classifier(t, act, s=s)
set = ClassifierSet(self)
```

Some similar modifications were required that meant obscuring the underlying algorithm, but were required to allow the framework to be flexible enough to meet requirements for its use (Requirements 5 and 6).

The second area that stands out as having suffered in the transition is the insertion and deletion functionality. As described in Section 9.4 (p. 43), this was required to deal with the various types of insertion and deletion that take place during the algorithm. The result is a flexible insertion and deletion system that should be capable of dealing with any future developments without major rework. It also succeeds in reducing the number of lines of code in the `deleteFromPopulation` procedure from 21 to 12, and in doing so adds clarity to the overall process. However, compared to the original algorithm or the procedural version, it requires a novice to the system to spend a greater amount of time to comprehend the subtleties of the different functions (i.e. the differences between `append`, `add` and `insert`).

One other criticism that could be leveled at the core structure is the use of class variables for the system parameters. This was reasoned in Section 9.5 (p. 49) and was a suitable mechanism to comply with Requirement 3. However the fact that the value of some of these parameters was initially `None` is not ideal.

Speed (Requirement 1b)

As Figure 13.1 shows, the object-oriented framework comes at a significant cost in running time. In fact, during these informal tests on the Boolean Multiplexer environment, the time is nearly double that of the procedural version (15.33 seconds compared to 8.62 seconds). The `Experiment` class, with and without report generation, was also compared against an equivalent program using the object-oriented framework directly. This showed a negligible difference in the time taken compared to when the `Experiment` class is used directly (15.33 seconds compared to 15.14 seconds for direct use), with an increase of around 3 seconds when the extra burden of reporting was added.

It should be noted at this point that these are not accurately measured timings and do not take into account background processes during the running period. However they were averaged over 90 experiments (3 runs of 30 experiments) and at the very least give the reader an idea of the relative timings.

Perhaps more significant is the comparison to Butz's Java implementation. The same experiment, including reporting, takes less than a second to run. To some extent this relative speed was expected due to the fact that Python is an interpreted language an has known performance issues in some areas.[2] However, it is somewhat disappointing that the Python framework should be so dramatically slower than its Java counterpart.

By analysing the code further it may be possible to identify some areas that would benefit from code optimisation. Using Python's `hotshot` module, it is possible to produce a profile of an experiment as it is executed. This records the number of function calls that were made and the relative time spent executing each one. These statistics can then be used to identify the functions that are called often and that occupy a significant proportion of processing time. The first 10 functions—when ordered by number of calls—are shown in the table below, see Appendix F for the full profile.

[2] See `http://www.python.org/moin/PythonSpeed/PerformanceTips` for more information.

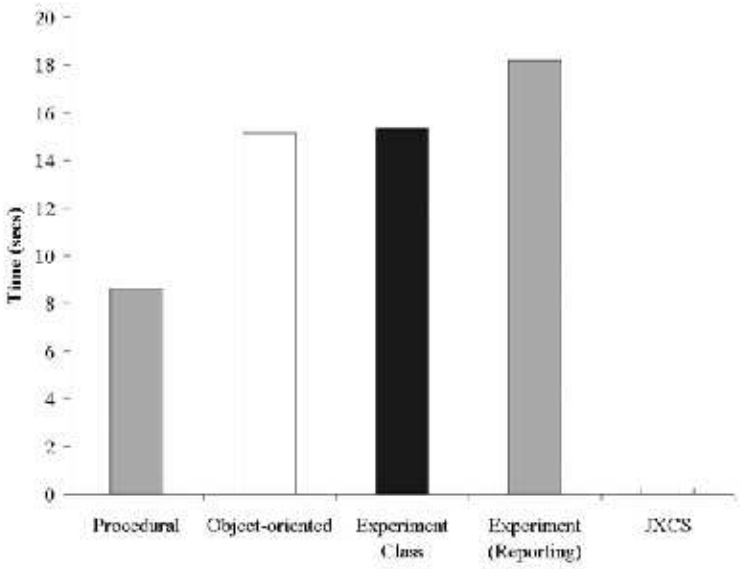

Figure 13.1: A comparison of the time taken to run 5000 exploit problems in the Boolean Multiplexer environment using five equivalent programs: (1) using the ProceduralXCS module, (2) using the object-oriented framework, (3) using the `Experiment` class, (4) using the `Experiment` class with reporting functionality enabled, and (5) using Butz's JXCS. Results are an average of 90 experiments (or 3 runs of 30 experiments).

# Calls	Class	Function	Time
863239	Classifier	doesMatch	10.316
462530	SelectionMechanisms	add	1.941
432142	Classifier	calculateDeletionVote	5.42
268383	Classifier	__ne__	1.664
268254	Classifier	__eq__	6.195
149154	ClassifierSet	append	0.645
130716	ClassifierSet	__iter__	1.228
118912	ClassifierSet	__len__	0.456
51066	ClassifierSet	__sum_attribute	7.123
40194	Classifier	couldSubsume	0.346
		Total Time	71.959

As the profile shows, the `doesMatch` method is called substantially more times than any of the other methods and occupies a large proportion of the execution time. Butz & Wilson (2000) describe the procedure thus:

```
for each attribute x in C_cl
    if(x ≠ # and x ≠ the corresponding attribute in σ)
        return false
return true
```

This was implemented in the procedural module using the most direct translation, shown below.[3] To run this method in isolation on 100 000 classifiers/situations took 9.467743 seconds (averaged over ten separate experiments).

```
for c, s_x in map(None, cl.C, s):
    if c != None and c != s_x: return False
return True
```

In an attempt to optimise this function, we could rewrite the function to access the list elements using an index. The code, shown below, took 9.261004 seconds under the same conditions as before.

```
for i in range(L):
    if cl.C[i] != None and cl.C[i] != s[i]: return False
return True
```

In a further attempt to reduce the computation time, we use a `while` loop. This was timed at 8.915548 seconds, a reduction of just under 6%.

```
i = 0
while i < len(cl.C):
    if cl.C[i] != None and cl.C[i] != s[i]: return False
    i += 1
return True
```

As the function is called so frequently in the algorithm, this final version was

[3] The `map` function is a built-in function that applies its first argument (usually a function) to the successive arguments. However it is used in this context to allow us to iterate through more than one list at a time.

used in the object-oriented framework in place of the more direct translation that was used in the procedural version.

This sort of optimisation is low-level and the lessons learnt can be replicated throughout the framework without much change to the overall algorithm. Less straightforward optimisation are higher-level design decisions, that also effect the readability of the code. For example, the `calculateDeletionVote` function is near the top of the function calls list; we could attempt to make the calculation of the deletion vote more efficient. One method would be to store the value of the vote and only recalculate it when the classifier's data is modified (as the deletion occurs in the entire population and only the Action Set classifiers are updated on each step). A similar technique could be applied to other scenarios such as the summation of attributes in the a set. The value could also be modified "on-the-fly"; keeping a running total or average of certain set attributes. For example, a total numerosity value is maintained by adding the value of new or subsumed classifiers and removing the value of old ones. Whether these solutions provide an improvement in speed or not will depend on the ratio of function calls to the rate of change in the dependant data. However, the introduction of this auxiliary code would certainly have an effect on the quantity of code in the framework and its overall legibility.

Therefore, considering the sharp contrast in execution time between this framework and the Java implementation, we consider this trade-off to feature heavily in any future development of the framework.

XCS Research and Development (Requirement 6)

It would be difficult to conclude with certainty that whether the framework does indeed provide a useful tool for research, as this will hopefully emerge once the framework is put to use. However, as anecdotal evidence here we show the implementation of the Tournament Selection modification described in Section 4.3 (p. 22). This is admittedly one of the more basic modifications that have been suggested, but it serves to highlight some of the issues involved.

Being a selection mechanism, we plan to augment the basic XCS functionality by subclassing the `SelectionMechanism` class to create a `Tournament` class with a specific constructor and the `add`, `select` and `reset` functionality of tournament selection:

```
from xcs.SelectionMechanisms import SelectionMechanism
class Tournament(SelectionMechanism):
    def __init__(self, tournament_size=0.4):
        [...]
    def add(self, weight, item):
        [...]
    def select(self):
        [...]
    def reset(self):
        [...]
```

The second step is to use this selection mechanism in an experiment. This

requires us to create an instance of the `ClassifierSet` class; set instances of the tournament selector for use during replication and deletion; then specify the set as an initial population for an experiment:

```
env = BooleanMultiplexer(2)
exp = Experiment(env)
cset = ClassifierSet(Classifier)
cset.replication_selector = Tournament()
cset.deletion_selector = Tournament()
exp.setInitialPopulation(cset)
exp.run()
```

Alternatively, the mechanism could be changed statically by subclassing the `ClassifierSet` to produce a class, such as `ClassifierSetTS`, that overrides `getDefaultDeletionSelector` and `getDefaultReplicationSelector` to return tournament selector objects rather than roulette wheels.

In both Butz's and Barry's Java implementation such an alteration would not be possible dynamically and would be arguably more difficult to achieve statically (via inheritance). It is more likely that a developer would modify the original source code as a means of inserting the functionality.

With regards to a modification such as classifier condition encoding, the use of lists for conditions means that the framework offers a very flexible basis for further development. For example, the list can store floats for continuous-valued inputs (Wilson 1999) or tuples for messy encoding (Lanzi 1999)[4]. However, as is the case with the other XCS implementations, there is still a tight coupling between the environment and the classifiers in this respect. To implement one of the aforementioned encodings, the developer would need to ensure that the environment generated situations in the format expected by the classifier conditions.

13.2 Future Work

The release of the first version of the framework concludes the current stage of development. There is, however, much scope for future development that was not possible within the scope of this project. This includes entirely new areas of functionality as well as improvements and enhancements to the existing framework.

Multi-step Envrionments

In this project we have mainly concentrated on the single-step Multiplexer problems. This was mainly due to the fact that the original algorithm was written mainly for single-step problems. Although the algorithm contains some functionality for multi-step problems, we were required to add more logic to our implementation in order to get the multi-step to work at all. However as the system testing showed, neither version of the algorithm is producing results with the same accuracy as for the single-step problems.

[4] See also Section 4.3 (p. 21)

Therefore the first step of any further development would be tackle this problem. One possibility is create separate `run` methods: one for single-step and one for multi-step problems. This was initially decided against as it was thought possible to run both problem types in the same code as was described in the algorithm. However this added complexity may be shielding the problems we are encountering with the multi-step problems.

Another way of tackling the problem would be to run the Python version against the Java version more exactly, using a seed classifier population and fixing the "random" selections so that both algorithms should behave *exactly* the same. In this way, the results could be analysed step-by-step and when differences occur the problem can be traced more accurately than simply by trying to analyse the performance statistics and final population.

Further Code Optimisation

Although it is perhaps over-optimistic to expect a reduction in execution time to that of the Java or C implementations, it is hoped that substantial gains could be made. In Section 5.3 (p. 26) three methods were proposed in order to obtain a fast execution time; code optimisation, using Python compilers and writing modules in C. It was beyond the scope of this project to explore much further into the effects of these techniques; so far we have only considered code optimisation, and this has not been an exhaustive search of the possibilities. We would expect that subsequent versions of the framework would focus heavily on this area to obtain a more satisfactory speed, without the loss of clarity.

Generating Statistics Graphics

The framework currently only outputs the statistics as a text file and relies on external scripts written for programs such as Gnuplot[5] to create graphic representations of the data.

One of the benefits of Python and other scripting languages is that it can be used to join together other processes and programs, even those written in different languages. There are several such packages that could be used to add the generation of the aforementioned statistics graphics. The `Gnuplot` package[6] is a set of Python modules that use Gnuplot directly and so can create any plot available in a standard Gnuplot package. Alternatively `SciPy`[7] is a stand-alone set of scientific tools for Python and is capable of a vast set of functions including data plotting and interpolation.

The use of these packages requires that the user has them installed on their system. Therefore we would not want to build such functionality into the core of the framework. However derivations of the `Experiment` class could be created that use one of these external libraries. This would allow the user to generate these plots quickly, easily and consistently.

[5] See `http://www.gnuplot.info/` for more information.
[6] See `http://gnuplot-py.sourceforge.net/` for more information.
[7] See `http://www.scipy.org/` for more information

GUI . . . and beyond!

Although it is not possible to remove the user entirely from the need to understand the algorithm and write code to implement their own environments and ideas, a graphical user interface could be of significant benefit to the framework. If a GUI were developed to enable the user to setup, run and analyse experiments it would allow the user to concentrate on actual XCS functionality rather than the application-specific code required to use the `Experiment` class.

Python contains several toolkits for producing such interfaces; of these the cross-platform Tk-based `Tkinter` package is the most widely used. It should be possible to build a graphical interface on top of the existing framework with few alterations required. The preliminary requirements of the experiment, such as initialising the environment and setting learning parameters, could be achieved relatively easily via a GUI. Assuming the it is running on a separate thread to the classifier system, the user interface can also register listener functions in the `Experiment` class to update its state and provide feedback during the experiment. With the addition of the relevant functions to the `Experiment` class some extra controls could also be added to pause and resume the algorithm during execution.

Python's introspection capabilities also generate a large range of possibilities for future development that ultimately lead towards a primitive IDE (Integrated Development Environment) in which the user develops their code through the user interface without directly creating or managing the Python files themselves. It should be said at this point that development of such a system is no small task. However, it does provide an example—if an extreme one—of the benefits that the Python framework could bring about.

13.3 Concluding Remarks

Although the final system testing revealed two major flaws in the framework – inaccuracy of multi-step problems and slow execution time – we believe the framework developed can still be a useful tool in XCS research. Especially if the issues regarding the multi-step environment can be solved.

In particular we can see a benefit as an educational tool for XCS. For a computer scientist who is new to XCS, the code in both the procedural and object-oriented version should be much easier to follow than that of either the Java or C implementations. Also, as it mirrors the description given by Butz & Wilson (2000), it can be used in combination with the paper to aid understanding of the algorithm.

In addition, the framework's flexible design and the accompanying documentation should make it relatively easy for a user with experience in XCS to use it for new research ideas. In fact it will probably be the use of the framework in the "real world" that will drive the direction of any future development and improvements.

Part V

Appendices

Appendix A

Sources of Existing Implementations

This study has focused mainly on the following XCS implementations for comparison purposes as they are the most commonly used implementations that are currently being used.

Alwyn Barry	Java	http://www.cs.bath.ac.uk/~amb/LCSWEB/jxcsawt.zip
	C	http://www.cs.bath.ac.uk/~amb/LCSWEB/xcsc.zip
Martin Butz	Java	ftp://ftp-illigal.ge.uiuc.edu/pub/src/XCSJava/XCSJava1.0.tar.Z
	C	ftp://ftp-illigal.ge.uiuc.edu/pub/src/XCS/XCS.tar.Z

Appendix B

Requirements Specification

The following is the requirements specification to which to the framework was developed. The sources of these requirements are documented and discussed in Chapter 4 (p. 19).

1. The code should be clear to read and understand for users new to XCS theory and/or the programming language used.
   (a) Where possible the code should emulate the algorithm described by Butz & Wilson (2000).
      i. Where changes have been made or extra functionality included, it should optional and fully documented.
   (b) Optimisations should be made to the algorithm where they give a clear speed advantage and do not affect Requirement 1a substantially.
   (c) The framework should be designed in an object-oriented structure.
   (d) The code should adhere to the common standards of the programming language (where this does not interfere with Requirement 1a).
   (e) The most recent stable version of the programming language should be used.
   (f) No external libraries or resources should be used that would require the user to install additional components.
2. The framework should be fully documented.
   (a) An automated documentation tool should be used to ensure uniform coverage and style.
   (b) Each function and module of the framework should be extensively commented.
   (c) Documentation should include installation and usage instructions.
   (d) Documentation should include both dynamic (e.g. hypertext) and static (e.g. PostScript) versions.
3. The system's learning parameters must be modifiable at run-time.

(a) Newly developed components should have access to these parameters.

(b) Newly developed components should be able to define their own parameters.

4. The framework should facilitate reporting of the data generated during an XCS experiment.

   (a) A report listing all the data about each classifier in the final population should be generated at the user's request.

   (b) A report listing moving averages of the system performance, system error and macroclassifier population during the experiment should be generated at the user's request.

      i. The user should be able to specify whether explore or exploit problems should be included in the report.

      ii. System performance in single-step problems should be calculated as the reward earned on each step.

      iii. System performance in multi-step problems should be calculated as the number of steps taken to solve the problem.

      iv. System error should be calculated as the difference between the predicted and the actual reward earned.

      v. The statistics should be reported as a fraction of their maximum values to allow them to use the same scale.

   (c) The user should be able to register functions to be fired at key events during an experiment.

      i. The user should be able to register functions to be executed at the end of each step.

      ii. The user should be able to register functions to be executed at the end of each problem.

      iii. The user should be able to register functions to be executed at the end of each experiment.

      iv. There should be no limit to the number of functions registered for each event.

      v. The functions should have access to all the current data in the XCS system.

5. The framework should facilitate the creation of new environments.

   (a) The environment's interface should be clearly defined for the user attempting to implement a new environment.

   (b) The user should only have to write the code to create the environment, no extra code should be required to apply it to the XCS algorithm.

6. The framework should facilitate the further development of the XCS algorithm.

   (a) The high-level operations of the algorithm (the main loop, action selection, GA and parent selection) should be modifiable dynamically.

(b) A logical class hierarchy should be used to enable fundamental changes (such as classifier condition encoding) to the system through subclassing.

Appendix C

Test Environments

There are two standard test environments often used in classifier system research and in this project. This section documents the "Boolean Multiplexer" and "Maze" environments in detail.

C.1 Boolean Multiplexer

A multiplexer has k input channels and 2^k data channels, with one output channel. The data channels are labelled with addresses from zero to $2^k - 1$ and the multiplexer chooses to output the data channel at the address given by the k input channels. Figure C.1 shows a circuit diagram of a multiplexer with two addressing channels ($k = 2$) and four data channels. As an example of its operation; if the address channel inputs were 0 and 0 respectively, the multiplexer would output the signal given by the first data channel (00).

With regards to the multiplexer problem for classifier systems; their task is to learn from an input message composed of the $k + 2^k$ binary inputs (the address channels plus the data channels), which output is correct. For example, if the address channels' values are 1 and 0, and the data channels are 0, 1, 1 and 0 respectively, then the classifier system is passed as an input string "100110". From this the classifier system should propose action 1 (the value of the third data channel) to receive a reward.

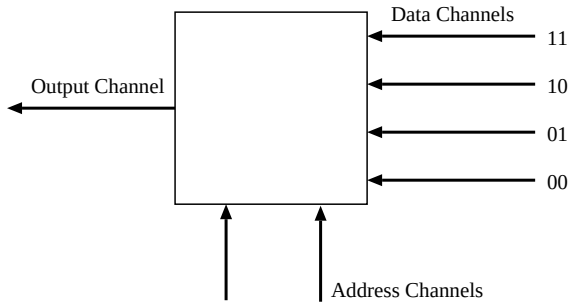

Figure C.1: A Boolean-6 Multiplexer

000### : 0 → 1000	000### : 1 → 0
001### : 1 → 1000	001### : 0 → 0
01#0## : 0 → 1000	01#0## : 1 → 0
01#1## : 1 → 1000	01#1## : 0 → 0
10##0# : 0 → 1000	10##0# : 1 → 0
10##1# : 1 → 1000	10##1# : 0 → 0
11###0 : 0 → 1000	11###0 : 1 → 0
11###1 : 1 → 1000	11###1 : 0 → 0

Figure C.2: The 16 optimal rules for a Boolean-6 multiplexer. To the left of the colon is the string representation of the inputs and immediately to the right is the proposed action. The value to the right of the arrow is the predicted reward.

The Boolean-6 Multiplexer is a relatively straight-forward, single-step problem for an XCS to solve. It is a reasonable expectation that it should have reached high performance (or low error) within 5000 repeatitions. After this time the population should be composed mainly of the 16 maximally general rules needed to succesfully classify the inputs. Figure C.2 shows these rules for a Boolean-6 Multiplexer with a reward of 1000 for correct classification. Larger multiplexers (11 and 20 inputs) are more complex common tests for classifier systems.

C.2 Mazes

The second class of environment are mazes. Wilson (1994) introduced the "Woods1" environment for the ZCS. This environment, and those that followed in XCS research, are known as *Class 1* environments because the system's next input and possible reward depends only on the current input and action, and no other historical data.[1]

Woods1 uses an infinitly repeating pattern of spaces, rocks and food to define a maze. In the maze is an "animat" (denoted by an "*") that is controlled by the classifier system. The animat is placed randomly in the maze and can move into any adjacent spaces not occupied by a rock. It receives reward for finding food, at which point the problem is reset and the animat begins from another random location. Figure C.3 shows a typical maze pattern.

The classifier system therefore receives sensor readings from the animat about the 8 adjacent squares. Each type of square is encoded into binary: "00" for a blank; "10" for a rock; and "11" for food. The animats 8 movements are translated into actions numbered 0 to 7 in the order: N, NE, E, SE, S, SW, W, NW.

A later maze introduced by Wilson (1995) adds a further intricacy to this basic problem. In"Woods2", an extra rock and food type are added. Although both food types and both rock types have the same consequences for the animat, it means that three bits are used to encode the sensor reading. This encoding is done in such a way that the right-most bit is not much use to the animat as it does not distinguish between rock and food. As a consequence the classifier system has more scope for generalisation of its condition strings.

[1] Or alternatively this is known as Markovian with delayed rewards.

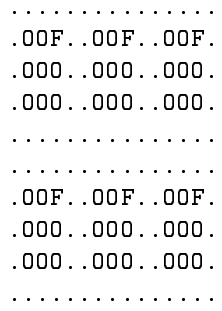

Figure C.3: A example of the Woods1 environment. It is usual to refer to blank spaces with a ".", rocks as "0" and food as "F".

A further environment, "Woods7", was also introduced for ZCS (Wilson 1994). It uses the same objects as Woods1 but has a much larger pattern that is more randomly distributed than Woods1. Woods7 is known as a Class 2 problem[2]. This increase in complexity is described by Wilson:

> [T]o know one's position in Woods1 (with respect to the basic configuration), it is sufficient to know the current input. In Woods7, however, it is necessary either to see more than one step away, or to remember some recent sensory inputs.

This second class of problem poses a more difficult challenge as sensor readings do not cover spaces more than one step away. Also ZCS/XCS systems do not have the use of temporary memory as other branches of classifier systems do. However, although it is expected that performance will drop substatially, Woods7 is often of interest simply to see how well it can do using simply its discounting reward algorithm.

[2] Or Non-Markovian with delayed rewards

Appendix D

Framework Schema

[Replace With Schema Page]

[Replace With Schema Page]

Appendix E

Experiment T-Tests

The t-tests carried out for the purposes of this project were two-tailed t-tests for unrelated samples with a significance level of 5automatically using a custom-built Python script to extract and analyse the data direct from the statistics output (see the program listing of `average.py` and `ttest.py` in Appendix G for more information).

The results from the t-test give a statistical judgement on whether the two populations being compared are the same or different. With 58 degrees of freedom (as there were 2 sets of 30 samples taken), the significance value at 5around 1.671. The tables below show the t values for each t-test, the figures in bold are the samples showing signficant difference (i.e. those greater than 1.671).

E.1 Boolean 6-Multiplexer

Procedural vs. JXCS			
Sample Point	Performance	Error	Population
250	0.583075	1.368201	**2.537301**
1000	**3.294280**	**3.685720**	**5.067799**
2000	**4.511831**	**4.098217**	**2.091658**
5000	**1.682043**	**6.319991**	1.375837

Object-oriented vs. JXCS			
Sample Point	Performance	Error	Population
250	1.414732	0.480466	**2.550073**
1000	0.574452	1.131931	1.214127
2000	**2.335014**	**2.724308**	0.445968
5000	**1.682043**	**6.613716**	**3.382524**

Procedural vs. Object-oriented			
Sample Point	Performance	Error	Population
250	0.877357	1.079390	0.723699
1000	1.536505	**1.735508**	**3.324798**
2000	1.157364	1.234692	**1.917258**
5000	0.000000	1.241375	0.232862

E.2 Woods2

Object-oriented vs. JXCS			
Sample Point	Performance	Error	Population
250	1.414732	0.480466	**2.550073**
1000	0.574452	1.131931	1.214127
2000	**2.335014**	**2.724308**	0.445968
5000	**1.682043**	**6.613716**	**3.382524**

Appendix F

Profile of an Experiment

# Calls	Class	Function	Time
863239	Classifier	doesMatch	10.316
462530	SelectionMechanisms	add	1.941
432142	Classifier	calculateDeletionVote	5.42
268383	Classifier	__ne__	1.664
268254	Classifier	__eq__	6.195
149154	ClassifierSet	append	0.645
130716	ClassifierSet	__iter__	1.228
118912	ClassifierSet	__len__	0.456
51066	ClassifierSet	__sum_attribute	7.123
40194	Classifier	couldSubsume	0.346
26036	ClassifierSet	__calculate_total_numerosity	3.999
24564	Classifier	updatePrediction	0.404
24564	Classifier	updateActionSetEstimate	0.372
24564	Classifier	updateFitness	0.328
19827	ClassifierSet	__init__	0.692
19826	Environment	__is_eop	0.14
16494	Classifier	isMoreGeneral	0.081
11670	Classifier	doesSubsume	0.387
11417	ClassifierSet	__get_super_set	0.533
11164	SelectionMechanisms	select	0.382
11164	SelectionMechanisms	reset	0.3
9943	ClassifierSet	__calculate_action_count	1.209
9914	Environment	__get_situation_length	0.083
9913	ClassifierSet	generateMatchSet	21.026
9913	ClassifierSet	generatePredictionArray	1.43
9913	Environment	__generateSituation	1.51
9913	ClassifierSet	generateActionSet	1.43
9913	XClassifierSystem	update	44.934
9913	XClassifierSystem	run	23.477
9913	ReinforcementProgram	getReward	0.588
9913	ClassifierSet	__calculate_average_prediction	3.903
9913	XClassifierSystem	selectAction	0.241

# Calls	Class	Function	Time
9913	ClassifierSet	__calculate_total_prediction	0.638
9913	Environment	reset	1.642
9913	Environment	executeAction	0.065
9913	XClassifierSystem	useExplore	0.061
9913	Environment	getSituation	0.06
6006	ClassifierSet	doDeletion	18.926
5960	ClassifierSet	selectOffspring	0.821
5960	Classifier	mutate	0.032
5598	ClassifierSet	insert	8.848
5204	ClassifierSet	__calculate_average_fitness	2.053
4913	ClassifierSet	update	2.293
4913	XClassifierSystem	runGA	35.914
4913	ClassifierSet	calculateTimeSinceGA	0.222
2980	Classifier	crossover	0.238
1514	ClassifierSet	add	0.062
479	ClassifierSet	remove	0.942
479	ClassifierSet	delete	0.953
46	ClassifierSet	addCoveringClassifier	0.011
46	Classifier	__init__	0.001
30	ClassifierSet	__get_unused_actions	0.001
6	SelectionMechanisms	__init__	0
1	XClassifierSystem	reset	0.001
1	XClassifierSystem	__init__	0.001
1	Environment	__get_no_actions	0
1	Environment	__get_max_steps	0
1	Environment	__get_max_reward	0
1	Experiment	run	71.959

Appendix G

Program Listing

The framework is organised into the following package for distribution. The source code for the core modules is shown in the following section.

- doc/
  - HTML Documentation
  - PostScript Documentation
- gnuplot/
  - Multiplexer GNUPlot script
  - Woods GNUPlot script
- src/
  - ProceduralXCS
  - XClassifierSystem
  - SelectionMechanisms
  - Experiment
  - test/
    - test_ProceduralXCS
    - test_XClassifierSystem
  - environments/
    - Environment
    - BooleanMultiplexer
    - Woods
- scripts/
  - Interactive Multiplexer Experiment
  - Interactive Woods Experiment
  - Statistics Averaging Script
  - T-test Script

- Read Me
- setup.py
- setup.exe

XClassifierSystem.py

Contains the object-oriented version of the core XCS algorithm.

Module Contents:

- **Classes**: XClassifierSystem, ClassifierSet, Classifier

[XClassifierSystem.py]

[XClassifierSystem.py]

[XClassifierSystem.py]

[XClassifierSystem.py]

[XClassifierSystem.py]

[XClassifierSystem.py]

[XClassifierSystem.py]

[XClassifierSystem.py]

[XClassifierSystem.py]

[XClassifierSystem.py]

ProceduralXCS.py

Contains the procedural version of the core XCS algorithm.

Module Contents:

- **Classes**: Classifier
- **Global Variables**: _env, _rp, _t, _params, _record
- **Functions**: setup, run, generateMatchSet, doesMatch, generateCoveringClassifier, generatePredictionArray, selectAction, generateActionSet, updateSet, updateFitness, runGA, selectOffspring, applyCrossover, applyMutation, actionSetSubsumeInsert, insertInPopulation, deleteFromPopulation, deletionVote, doActionSetSubsumption, doesSubsume, couldSubsume, isMoreGeneral, reset, output_stats, number_of_actions, get_environment, set_environment, get_reinforcement_program, set_reinforcement_program, get_timestep, set_timestep, get_parameter, set_parameter.

[ProceduralXCS.py]

[ProceduralXCS.py]

[ProceduralXCS.py]

[ProceduralXCS.py]

[ProceduralXCS.py]

[ProceduralXCS.py]

[ProceduralXCS.py]

[ProceduralXCS.py]

[ProceduralXCS.py]

Environment.py

Contains abstract classes that define the environment external to the classifier system.

Module Contents:

- **Classes**: Environment, ReinforcementProgram

[Environment.py]

[Environment.py]

SelectionMechanism.py

Contains the abstract class for a selection mechanism and two concrete classes that implement roulette wheel and tournament selection.

Module Contents:

- **Classes**: SelectionMechanism, RouletteWheel, Tournament

[SelectionMechanism.py]

[SelectionMechanism.py]

[SelectionMechanism.py]

BooleanMultiplexer.py

Contains a concrete derivation of the Environment and Reinforcement classes
to define the Boolean Multiplexer problem.

Module Contents:

- **Classes**: BooleanMultiplexer

[BooleanMultiplexer.py]

[BooleanMultiplexer.py]

Woods.py

Contains a concrete derivation of the Environment and Reinforcement classes
to define the Woods problem.

Module Contents:

- **Classes**: Woods

[Woods.py]

[Woods.py]

[Woods.py]

average.py

Averages out a text file containing XCS statistics taken from numerous experiments.

[average.py]

ttest.py

Calculates a t-test for the samples in the two specified files and works out if they are significantly different at the specified significance level.

[average.py]

Bibliography

Abbott, R. J. (1983), 'Program design by informal english descriptions', *Communications of the ACM* **26**(11), 882–894.

Barry, A. (2000), XCS Performance and Population Structure within Multiple-Step Environments, PhD thesis, Queens University Belfast.

Barry, A., Holmes, J. & Llora, X. (2004), Data mining using learning classifier systems, *in* L. Bull, ed., 'Applications of Learning Classifier Systems', Springer-Verlag.

Booker, L. B. (1989), Triggered rule discovery in classifier systems, pp. 265–274.

Butz, M. V. & Wilson, S. W. (2000), An algorithmic description of xcs, Technical Report 2000017, Illinois Genetic Algorithms Laboratory.

Butz, M. V., Sastry, K. & Goldberg, D. E. (2003), Tournament selection in xcs, pp. 1857–1869.

Carse, B., Fogarty, T. C. & Munro, A. (1996), Evolving temporal fuzzy rule-bases for distributed routing control in telecommunication networks, *in* F. Herrera & J. Verdegay, eds, 'Studies in Fuzziness and Soft Computing', Physica-Verlag, pp. 467–488.

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1994), *Design Patterns*, Addison-Wesley.

Hearst, M. A. & Hirsh, H. (2002), 'Ai's greatest trends and controversies'. `http://www.computer.org/intelligent/articles/AI_controversies.htm`.

Holland, J. H. (1986), 'A mathematical framework for studying learning in classifier systems', *Physica D* **22**, 307–317.

Lanzi, P. L. (1999), Extending the representation of classifier conditions part i: From binary to messy coding, pp. 337–344.

Lundh, F. (2005), 'Python performance tips'. `http://www.python.org/moin/PythonSpeed/PerformanceTips`.

van Rossum, G. (2005), 'Python patterns – an optimization anecdote'. `http://www.python.org/doc/essays/list2str.html`.

Widrow, B. & Hoff, M. E. (1960), 'Adaptive switching circuits', *WESCON Convention Record Part 4* pp. 96–104.

Wilson, S. W. (1985), Knowledge growth in an artificial animal, pp. 16–23. Also appeared in Proceedings of the 4th Yale.

Wilson, S. W. (1994), 'Zcs: A zeroth level classifier system', *Evolutionary Computation* **2**(1), 1–18.

Wilson, S. W. (1995), 'Classifier fitness based on accuracy', *Evolutionary Computation* **3**(2), 149–175.

Wilson, S. W. (1998), Generalization in the xcs classifier system, pp. 665–674.

Wilson, S. W. (1999), Get real! xcs with continuous-valued inputs, *in* L. Booker, S. Forrest, M. Mitchell & R. L. Riolo, eds, 'Festschrift in Honor of John H. Holland', Center for the Study of Complex Systems, pp. 111–121.